

第一章 计算机系统结构的基本概念

第一台电子计算机问世已经半个世纪了,计算机已经历了五次更新换代。第一代计算机(1945—1954)将电子管和继电器存储器用绝缘导线互连在一起,由单个 CPU 构成,CPU 用程序计数器和累加器顺序完成定点运算,采用机器语言或汇编语言,用 CPU 程序控制 I/O。代表性系统有由 John Von Neumann, Arthur Burks 和 Herman Goldstine 于 1946 年在普林斯顿研制成功的 IAS 计算机、由宾夕法尼亚大学莫尔学院于 1950 年制成的 ENIAC、由 IBM 于 1953 年制造的 IBM701 计算机。第二代计算机(1955—1964)采用分立式晶体三极管、二极管和铁氧体的磁芯,用印刷电路将它们互连起来。采用了变址寄存器、浮点运算、多路存储器和 I/O 处理机。采用有编译程序的高级语言、子程序库、批处理监控程序。代表性系统有 1959 年制成的 Univac LARC、20 世纪 60 年代的 CDC1604 和 1962 年制成的 IBM7030。第三代计算机(1965—1974)采用小规模或中规模集成电路和多层印刷电路。微程序控制在这一代开始普及。采用了流水线、高速缓存和先行处理机。软件方面采用多道程序设计和分时操作系统。代表性系统有 IBM/360-370 系列、CDC6600/7600 系列、Texas 仪表公司的 ASC、Digital Equipment 公司的 PDP-8 系列。第四代计算机(1974—1991)采用大规模或超大规模集成电路和半导体存储器。出现了用共享存储器、分布存储器或向量硬件选择的不同结构的并行计算机,开发了用于并行处理的多处理操作系统、专用语言和编译器,同时产生了用于并行处理或分布处理的软件工具和环境。代表性系统有 VAX9000、CrayX-MP、IBM/3090VF、BBNTC-2000 等。第五代计算机(1991—现在)采用 VLSI 工艺更加完善的高密度、高速度处理机和存储器芯片。它的最重要特点是进行大规模并行处理,采用可扩展的和容许时延的系统结构。代表性系统有 Fujitsu 的 VPP500、Cray Research 的 MPP、Thinking Machines 公司的 CM-5、Intel 超级计算机系统 Paragon、SGI 的 Origin 2000 和 Sun 公司的 10000 服务器。前三代中,每一代持续大约十年,第四代的时间跨度约为十五年。现在进入第五代。我们可以看到,换代的标志主要有两个:第一是计算机的器件。器件发生了根本的变化,经电子管、晶体管发展到集成电路,而集成电路又由小规模、中规模,到大规模和超大规模的阶段。器件的更新,其速度、功能、可靠性的不断提高和成本的不断降低,是计算机发展的物质基础。因此,器件的换代是计算机换代的最突出的标志。第二是系统结构的特点。系统结构不断改进,许多重要概念不断提出并且得到实现。例如变址寄存器概念、通用寄存器概念、浮点数据表示概念、程序中断概念、输入输出通道概念、间接寻址概念、虚拟存储器概念、Cache 存储器概念、系列化概念、微程序设计技术等。很明显,如果用大规模集成电路实现早期的计算机系统结构,人们并不会认为它是第四代计算机。因为第四代计算机在系统结构上较之早期的计算机已经有很大的改进和发展。因此,系统结构方面的特点同样是计算机换代的重要标志。

回顾计算机的发展历史,可以看出,计算机系统性能的不断提

系统结构的改进。恩斯洛(P. H. Enslow)曾经比较了1965—1975这十年间,器件延迟时间和计算机指令时间的关系。结果表明,这十年间器件延迟时间降低至原来的十分之一,但计算机的指令时间却是原来指令时间的百分之一。这种情况在计算机近年来的发展中变得更加明显。如何最合理地利用新器件,最大限度地发挥其潜力,设计并构成综合性能指标最佳的计算机系统,单纯依靠器件变革是不能解决的,还要靠计算机系统结构上的改进。

本书专门研究计算机系统结构,特别是高性能计算机系统结构。重点在于系统结构的设计和分析。

那么,什么是计算机系统结构?计算机系统结构和技术有什么关系?计算机系统结构的评价标准是什么?高性能计算机系统的技术是什么?本章就是要解决这些问题。

1.1 计算机系统结构

1.1.1 计算机系统层次结构

计算机系统由硬件/器件和软件组成,按功能划分成多级层次结构,如图1.1所示。图中每一级各对应一种机器,其作用和组成如图1.2所示。在这里,“机器”只对一定的观察者而存在。它的功能体现在广义语言上,能对该语言提供解释手段,如同一个解释器,然后作用在信息处理和控制对象上。在某一层次的观察者看来,他只是通过该层次的语言来了解和使用计算机,不必关心再内层的那些机器是如何工作和如何实现各自功能的。

图1.1中的第0级机器由硬件实现,第1级机器由微程序(固件)实现,第2级至第6级机器由软件实现。我们称由软件实现的机器为虚拟机器,以区别于由硬件或固件实现的实际机器。

第0级和第1级是具体实现机器指定功能的中央控制部分。它根据各种指令操作所需要的控制时序,配备一套微指令,编写出微程序,控制信息在各寄存器之间的传送,这就是第1级机器。实现这些微指令本身的控制时序只需要很少的逻辑线路,可采用硬联逻辑实现,它就是第0级机器,是机器的硬件内核。

第2级是传统机器语言机器。这级的机器语言是该机的指令系统。机器语言程序员用这级指令系统编写的程序由第1级的微程序进行解释。

第3级是操作系统机器。这级的机器语言中的多数指令是传统机器的指令,如算术运算、逻辑运算和移位等指令。此外,这一级还提供操作系统级指令,例如打开文件、读/写文件、关闭文件等指令。用这一级语言编写的程序,即那些与第2级指令相同的指令直接由微程序实现。操作系统级指令部分由操作系统进行解释。操作系统是运行在第2级上的解释程序。

第4级是汇编语言机器。这级的机器语言是汇编语言。用汇编语言编写的程序首先翻译成第3级或第2级语言,然后再由相应的机器进行解释。完成翻译的程序叫做汇编程序。

第5级是高级语言机器。这级的机器语言就是各种高级语言。用这些语言所编写的程序一般是由编译程序翻译到第4级或第3级上的语言,个别的高级语言也用解释的方

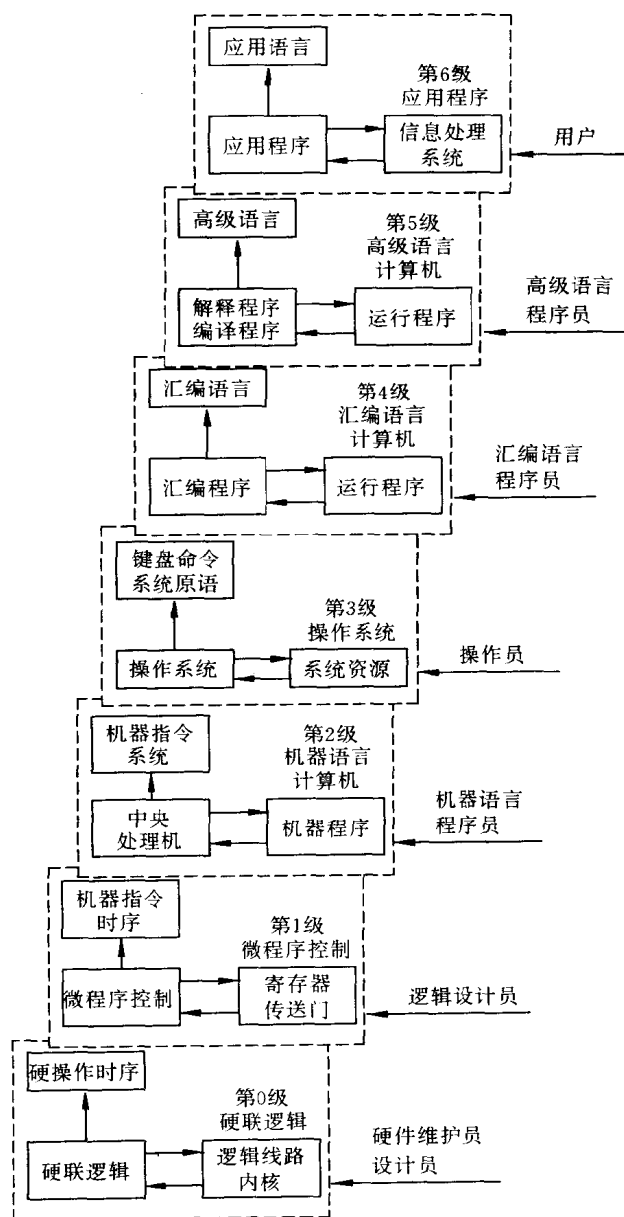


图 1.1 计算机系统层次结构

法实现。

第6级是应用语言机器。这级的机器语言是应用语言。这种语言使非计算机专业人员也能直接使用计算机，只需在用户终端用键盘或其他方式发出服务请求就能进入第6级的信息处理系统。

从学科领域来划分，大致可以认为第0至第1级是计算机组织与结构讨论的范围，第3至第5级是系统软件，第6级是应用软件。但是，严格说起来又不尽然，它们之间仍有交

叉。例如,第0级要求一定的数字逻辑基础;第2级涉及汇编语言程序设计的内容;第3级与计算机系统结构密切相关。在特殊的计算机系统中,有些级别可能不存在。

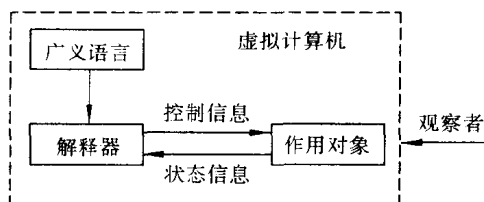


图 1.2 机器的作用和组成

虚拟机器新的实现方法,设计新的计算机系统。

把计算机系统按功能划分成多级层次结构,首先有利于正确地理解计算机系统的工作,明确软件、硬件和固件在计算机系统中的地位 and 作用。其次有利于理解各种语言的实质及其实现。最后还有利于探索

1.1.2 计算机系统结构定义

“计算机系统结构”这个名词来源于英文 computer architecture,也有译成“计算机体系结构”的。architecture 这个字原来用于建筑领域,其意义是“建筑学”、“建筑物的设计或式样”,它是指一个系统的外貌。60 年代这个名词被引入计算机领域,“计算机系统结构”一词已经得到普遍应用,它研究的内容不但涉及计算机硬件,也涉及计算机软件,已成为一门学科。但对“计算机系统结构”一词的含义仍有多种说法,并无统一的定义。

计算机系统结构这个词是 Amdahl 等人在 1964 年提出的。他们把系统结构定义为由程序设计者所看到的一个计算机系统的属性,即概念性结构和功能特性。这实际上是计算机系统的外特性。按照计算机层次结构,不同程序设计者所看到的计算机有不同的属性。使用高级语言的程序员所看到的计算机属性主要是软件子系统和固件子系统的属性,包括程序语言以及操作系统、数据库管理系统、网络软件等用户界面。Amdahl 等人提出的系统结构定义中的程序设计者是指为机器语言或编译程序设计者所看到的计算机属性,是硬件子系统的概念结构及其功能特性,包括机器内的数据表示,即硬件能直接辨认和处理的那些数据类型;寻址方式,包括最小寻址单元和地址运算等;寄存器定义,包括操作数寄存器、变址寄存器、控制寄存器等的定义、数量和使用方式;指令系统,包括机器指令的操作类型和格式、指令间的排序和控制机构等;中断机构,包括中断的类型和中断响应硬件的功能等;机器工作状态的定义和切换,如管态和目态等;输入输出结构,包括输入输出的连结方式,处理机/存储器与输入输出设备间数据传送的方式和格式、传送的数据量、以及输入输出操作的结束与出错标志等;信息保护,包括信息保护方式和硬件对信息保护的支持等等。这些即是程序员为了使其所编写的程序能在机器上正确运行,需要了解和遵循的计算机属性。当然不包括基本的数据流、控制流、逻辑设计和物理实现等。

在计算机技术中,一种本来是存在的事物或属性,但从某种角度看似乎不存在,称为透明性现象。通常,在一个计算机系统中,低层机器级的概念性结构和功能特性,对高级语言程序员来说是透明的。由此看出,在层次结构的各个级上都有它的系统结构。

计算机系统结构作为一门学科,主要研究软件、硬件功能分配和对软件、硬件界面的确定,即哪些功能由软件完成,哪些功能由硬件完成。

关于计算机系统结构这一概念,至今有各种各样的理解,很难有一个通用的定义。在下节讨论计算机组成和实现后,我们还要给出另一些定义。

1.1.3 计算机组成与实现

计算机组成的任务是在计算机系统结构确定分配给硬件子系统的功能及其概念结构之后,研究各组成部分的内部构造和相互联系,以实现机器指令级的各种功能和特性。这种相互联系包括各功能部件的配置、相互连接和相互作用。各功能部件的性能参数相互匹配,是计算机组成合理的重要标志,因而相应地就有许多计算机组织方法。例如,为了使存储器的容量大、速度快,人们研究出层次存储系统和虚拟存储技术。在层次存储系统中,又有高速缓存、多模块交错工作、多寄存器组和堆栈等技术。为了使输入输出设备与处理机间的信息流量达到平衡,人们研究出通道、外围处理机等方式。为了提高处理机速度,人们研究出先行控制、流水线、多执行部件等方式。在各功能部件的内部结构研究方面,产生了许多组合逻辑、时序逻辑的高效设计方法和结构。例如,在运算器方面,出现了多种自动调度算法和结构等。

由此可见,计算机组成是计算机系统结构的逻辑实现,包括机器内部的数据流和控制流的组成以及逻辑设计等。计算机组成的设计是按所希望达到的性能价格比,最佳、最合理地把各种设备和部件组成计算机,以实现所确定的计算机系统结构。一般计算机组成设计包括数据通路宽度的确定、各种操作对功能部件的共享程度的确定、专用功能部件的确定、功能部件的并行性确定、缓冲器和排队的确定、控制机构的设计、可靠性技术的确定等。对传统机器程序员来说,计算机组成的设计内容一般是透明的。

计算机实现是指计算机组成的物理实现。它包括处理机、主存等部件的物理结构,器件的集成度和速度,信号传输,器件、模块、插件、底板的划分与连接,专用器件的设计,电源、冷却、装配等技术以及有关的制造技术和工艺等。

计算机系统结构、计算机组成和计算机实现是三个不同的概念。系统结构是计算机系统的软、硬件的界面;计算机组成是计算机系统结构的逻辑实现;计算机实现是计算机组成的物理实现。它们各自包含不同的内容,但又有紧密的关系。

我们还应看到系统结构、组成和实现所包含的具体内容是随不同机器而变化的。有些计算机系统是作为系统结构的内容,其他计算机系统可能是作为组成和实现的内容。开始是作为组成和实现提出来的设计思想,到后来就可能被引入系统结构中。例如高速缓冲存储器一般是作为组成提出来的,其中存放的信息全部由硬件自动管理,对程序员来说是透明的。然而,有的机器为了提高其使用效率,设置了高速缓冲存储器的管理指令,使程序员能参与高速缓冲存储器的管理。这样,高速缓冲存储器又成为系统结构的一部分,对程序员来说是不透明的。

Amdahl 等人的计算机系统结构定义的主要内容是指令系统及其执行模型。根据这个定义,一个系列机中不同档次的机器有相同的系统结构。Amdahl 等人定义系统结构时认为只要指令系统兼容就能保证程序正确运行。由于程序的执行要依赖于程序库、操作系统和其他 Amdahl 等人的系统结构定义中没有涉及的因素,这要求操作系统接口等其他层次的标准化。同时,由于 VLSI 的迅速发展及其成本急剧下降,有些系列机推出有新指令的机器,例如 24 位地址的 IBM360 和 370 系统发展为 31 位地址的 370xA 系统,16 位地址的 PDP-11 发展为 32 位地址的 VAX 系列。随着新器件的出现,当今计算机设计者面

临的问题与 10 年前面临的问题大不相同,所以我们应当把计算机系统结构定义得更宽一些,除了 Amdahl 等人定义的内容外,还应包括功能模块的设计。也就是说,计算机系统结构、计算机组成、计算机实现之间的界限越来越模糊了。

1.1.4 计算机系统结构的分类

研究计算机系统分类方法有助于人们认识计算机的系统结构和组成的特点,理解系统的工作原理和性能。

通常把计算机系统按其性能与价格的综合指标分为巨型、大型、中型、小型、微型等。但是,随着技术的不断进步,各种型号的计算机性能指标都在不断进步,以至于过去的一台大型计算机的性能甚至于还比不上今天的一台微型计算机,而用过去一台大型计算机的价钱,今天却能够买一台性能指标高许多倍的新式大型计算机。可见按巨、大、中、小、微来划分的绝对性能标准是随时间变化而变化的。

计算机系统还可以根据其面向应用领域的不同性质而进行分类。一般说来,计算机都是作为通用系统进行设计的,但是,在用户编写应用程序时,却都带有专用性质。为了解决这个矛盾,采取的办法有:灵活地改变系统配置,包括内存容量、外围设备品种和数量等;允许适应特殊环境的要求采取不同的物理安装;增加处理不同数据结构的能力,如浮点、字符串、快速傅里叶变换等;提供多种可用的语言和操作系统,以适应批处理、分时、实时、事务处理等不同需要。所以按用途分类可以分为科学计算、事务处理、实时控制、家用等计算机。

按处理机个数和种类分,计算机系统又可分为单处理机、多处理机、并行处理机、关联处理机、超标量处理机、超流水线处理机、SMP(对称多处理机)、MPP(大规模并行处理机)、机群系统等。

下面再介绍三种常用的分类方法。

1. Flynn 分类法

1966 年 M. J. Flynn 提出了如下定义:

指令流(instruction stream)——机器执行的指令序列。

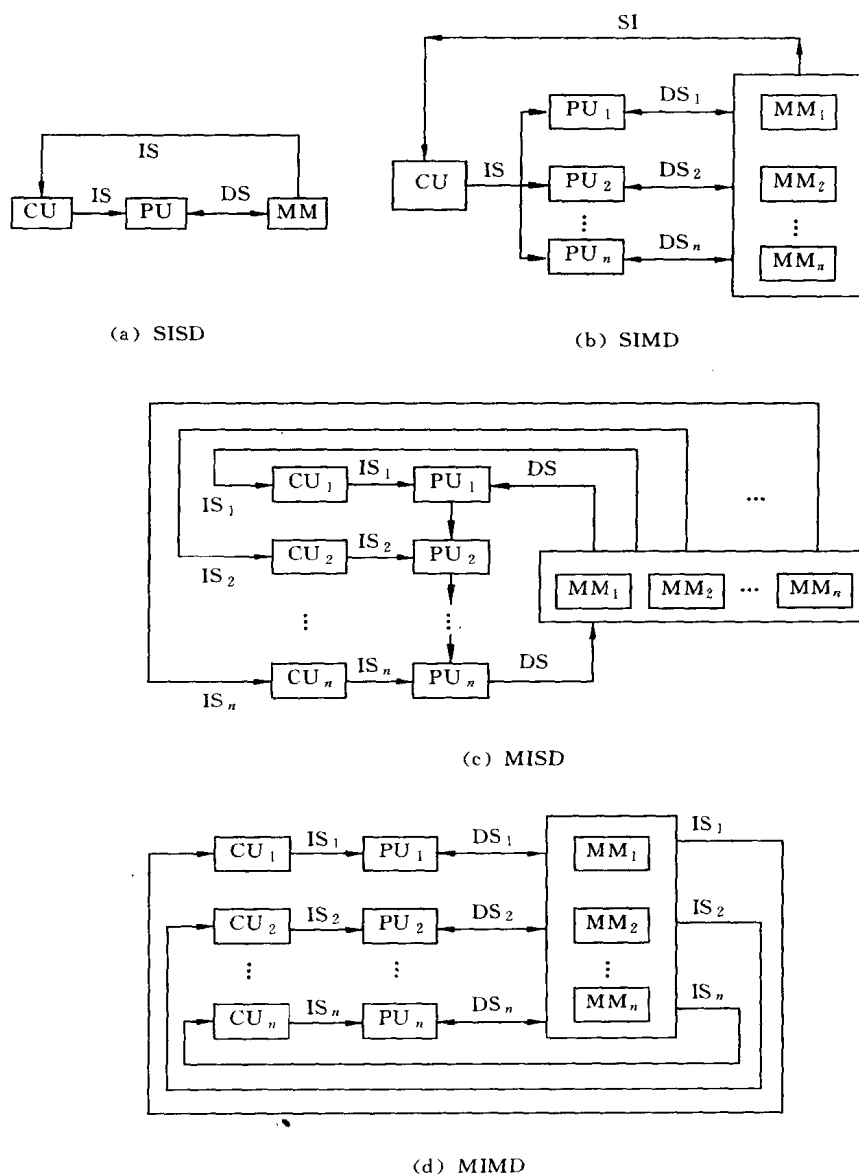
数据流(data stream)——由指令流调用的数据序列,包括输入数据和中间结果。

多倍性(multiplicity)——在系统最受限制的元件上同时处于同一执行阶段的指令或数据的最大可能个数。

同时,他按照指令流和数据流的不同组织方式,把计算机系统的结构分为以下四类:

- (1) 单指令流单数据流 SISD(Single Instruction stream Single Datastream)
- (2) 单指令流多数据流 SIMD(Single Instruction Stream Multiple Datastream)
- (3) 多指令流单数据流 MISD(Multiple Instruction Stream Single Datastream)
- (4) 多指令流多数据流 MIMD(Multiple Instruction Stream Multiple Datastream)

对应于这四类计算机的基本结构框图如图 1.3 所示。SISD 是传统的顺序处理计算机。SIMD 以阵列处理机或并行处理机为代表。MISD 在实际上代表何种计算机,也存在着不同的看法,有的文献把流水线结构机器看作是 MISD 结构。多处理机属于 MIMD 结构。



CU: 控制部件 PU: 处理部件 MM: 存储器模块 IS: 指令流 DS: 数据流

图 1.3 Flynn 分类法各类机器结构

2. 冯氏分类法

冯泽云于 1972 年提出用最大并行度对计算机系统结构进行分类。最大并行度 P_m 定义为：计算机系统在单位时间内能够处理的最大的二进制位数。假定每个时钟周期 Δt_i 内

能同时处理的二进制位数为 P_i ，则 T 个时钟周期内平均并行度为 $P_a = \frac{\sum_{i=1}^T P_i \Delta t_i}{T}$ 。平均并行

度不同于最大并行度,它取决于系统的运用程度,与应用程序有关。因此,定义系统在周期

$$T \text{ 内的平均利用率为 } \mu = \frac{P_a}{P_m} = \frac{\sum_{i=1}^T P_i}{TP_m}。$$

图 1.4 给出用最大并行度对计算机系统结构进行分类的方法。用平面直角坐标系中的一点代表一个计算机系统,横坐标代表字宽(n 位),即在一个字中同时处理的二进位的位数;纵坐标代表位片宽度(m 位),即在一个位片中能同时处理的字数。于是,一个系统的最大并行度就可以用这两个量的乘积,即用通过该点的水平线和垂直线与两坐标轴围成的矩形面积来表示。

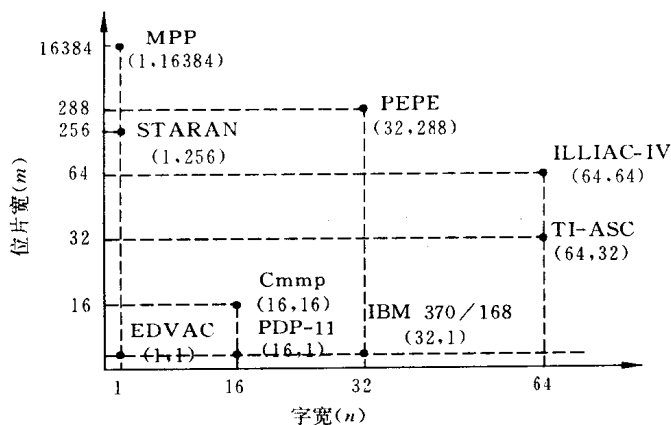


图 1.4 按最大并行度的冯氏分类法

由图 1.4 可得出四类不同处理方法的计算机系统结构:

(1) 字串位串 WSBS(word serial and bit serial),其中 $n=1, m=1$ 。这是第一代计算机发展初期的纯串行计算机。

(2) 字并位串 WPBS(word parallel and bit serial),其中 $n>1, m=1$ 。这是传统并行单处理机。

(3) 字串位并 WSBP(word serial and bit parallel),其中 $n=1, m>1$ 。STARAN, MPP, DAP 属于这种结构。

(4) 字并位并 WPBP(word parallel and bit parallel),其中 $n>1, m>1$ 。PEPE, ILLIAC IV, Cmp 属于这种结构。

3. Händler 分类法

Wolfgang Händler 在 1977 年根据并行度和流水线提出了另一种分类法。这种分类方法把计算机的硬件结构分成三个层次,并分别考虑它们的可并行-流水处理程度。这三个层次是:

- (1) 程序控制部件(PCU)的个数 k ;
- (2) 算术逻辑部件(ALU)或处理部件(PE)的个数 d ;
- (3) 每个算术逻辑部件包含基本逻辑线路(ELC)的套数 w 。

这样我们可以把一个计算机系统的结构用如下公式表示： $t(\text{系统型号}) = (k, d, w)$ 。为了进一步揭示流水线的特殊性，一个计算机系统的结构可用如下公式表示： $t(\text{系统型号}) = (k \times k', d \times d', w \times w')$ ，其中： k' 表示宏流水线中程序控制部件的个数， d' 表示指令流水线中算术逻辑部件的个数， w' 表示操作流水线中基本逻辑线路的套数。

例如 Cray1 有 1 个 CPU，12 个相当于 ALU 或 PE 的处理部件，可以最多实现 8 级流水线。字长为 64 位，可以实现 1~14 位流水线处理。所以 Cray1 的系统结构可表示为：

$$t(\text{Cray1}) = (1, 12 \times 8, 64(1 \sim 14))$$

下面是用这种分类法的例子：

$$t(\text{PDP11}) = (1, 1, 16)$$

$$t(\text{ILLIAC IV}) = (1, 64, 64)$$

$$t(\text{STARAN}) = (1, 8192, 1)$$

$$t(\text{Cmmp}) = (16, 1, 16)$$

$$t(\text{PEPE}) = (1 \times 3, 288, 32)$$

$$t(\text{TIASC}) = (1, 4, 64 \times 8)$$

1.2 计算机系统设计技术

1.2.1 计算机系统设计的定量原理

下面介绍计算机系统设计中经常用到的几个定量原理。

1. 加快经常性事件的速度 (make the common case fast)

这是计算机设计中最重要也最广泛采用的设计准则。使经常性事件的处理速度加快能明显提高整个系统的性能。一般说来，经常性事件的处理比较简单，因此比不经常出现的事件处理起来要快。例如，在 CPU 中两个数进行相加运算时，相加结果可能出现溢出现象，也可能无溢出发生，显然经常出现的事件是不发生溢出的情况，而溢出是偶然发生的事件。因此，在设计时应优化不发生溢出的情况，使这个经常性事件的处理速度尽可能快，而对溢出处理则不必过多考虑优化。因为发生溢出的概率很小，即使发生了，处理得慢一些也不会对系统性能产生很大的影响。

在计算机设计中经常会遇到上述情况。那么，如何确定经常性事件以及如何加快处理它，这就是下面介绍的 Amdahl 定律要解决的问题。

2. Amdahl 定律

Amdahl 定律告诉我们：系统中某一部件由于采用某种更快的执行方式后整个系统性能的提高与这种执行方式的使用频率或占总执行时间的比例有关。Amdahl 定律定义了由于采用特殊的方法所能获得的加速比的大小。

$$\text{加速比} = (\text{采用改进措施后的性能}) / (\text{没有采用改进措施前的性能})$$

$$= (\text{没有采用改进措施前执行某任务的时间}) / (\text{采用改进措施后执行某任务的时间})$$

Amdahl 定律中，加速比与两个因素有关：一个是计算机执行某个任务的总时间中可被改进部分的时间所占的百分比，即(可改进部分占用的时间)/(改进前整个任务的执行

时间),记为 Fe ,它总小于 1。另一个是改进部分采用改进措施后比没有采用改进措施前性能提高倍数,即(改进前改进部分的执行时间)/(改进后改进部分的执行时间),记为 Se ,它总大于 1。

我们可以得出如下结论:

(1) 改进后整个任务的执行时间为:

$$T_n = T_0 \left(1 - Fe + \frac{Fe}{Se} \right)$$

其中 T_0 为改进前的整个任务的执行时间。

(2) 改进后整个系统的加速比为

$$S_n = \frac{T_0}{T_n} = \frac{1}{(1 - Fe) + Fe/Se}$$

上面式子中 $(1 - Fe)$ 表示不可改进部分,显然当 Fe 为 0,即没有可改进部分时, S_n 为 1,所以性能的提高幅度受改进部分所占比例的限制。当 $Se \rightarrow \infty$ 时,则 $S_n = \frac{1}{1 - Fe}$,因此,可获取性能改善极限值受 Fe 值的约束。

下面举几个例子来说明 Amdahl 定律的应用。

例 1.1 假设将某系统的某一部件的处理速度加快到 10 倍,但该部件的原处理时间仅为整个运行时间的 40%,则采用加快措施后能使整个系统的性能提高多少?

解: 由题意可知: $Fe = 0.4$, $Se = 10$,根据 Amdahl 定律, $S_n = \frac{1}{0.6 + 0.4/10} = \frac{1}{0.64} \approx 1.56$

Amdahl 定律可以告诉我们一项改进措施可以使整个系统的性能提高多少,同时还能告诉我们为了改进性能价格比,如何合理分配系统的资源。

例 1.2 采用哪种实现技术来求浮点数平方根 FPSQR 的操作对系统的性能影响较大。假设 FPSQR 操作占整个测试程序执行时间的 20%。一种实现方法是采用 FPSQR 硬件,使 FPSQR 操作的速度加快到 10 倍。另一种实现方法是使所有浮点数据指令的速度加快,使 FP 指令的速度加快到 2 倍,还假设 FP 指令占整个执行时间的 50%。请比较这两种设计方案。

解: 分别计算出这两种设计方案所能得到的加速比:

$$S_{FPSQR} = \frac{1}{(1 - 0.2) + 0.2/10} = \frac{1}{0.82} = 1.22$$

$$S_{FP} = \frac{1}{(1 - 0.5) + 0.5/2} = \frac{1}{0.75} = 1.33$$

可见使所有 FP 指令的速度提高这一方案更好。

3. CPU 性能公式

在前面我们利用 Amdahl 定律比较两种改进 FPSQR 操作的效果时,需要知道 FPSQR 硬件方法和改进 FP 操作的时间,有时候要直接测出这些时间比较困难。这里我们介绍 CPU 性能公式方法,即把 CPU 的执行时间分解成三个独立的部分。如果我们能知道某种方案对这三部分是如何影响的,那么就能知道采用这种方案时整个系统的性能了。

我们知道大多数计算机的时钟速度是固定的,它的运行周期称为时钟周期。时钟可以

用时钟周期的长度(如 2ns)或其频率(如 500MHz)来表示。一个程序所花的 CPU 时间可以用两种方式来表示:

$$\text{CPU 时间} = \text{CPU 时钟周期数} / \text{频率}$$

或

$$\text{CPU 时间} = \text{CPU 时钟周期数} \times \text{时钟周期长}$$

除了用时钟周期数来表示一个程序的执行时间外,还可以用指令的条数(IC)来表示一个程序的执行时间。如果我们知道了执行的指令条数和所用的时钟周期数目,就可以算出每条指令的平均时钟周期数 CPI:

$$\text{CPI} = \text{CPU 时钟周期数目} / \text{IC}$$

代换可得: $\text{CPU 时间} = \text{IC} \times \text{CPI} \times \text{时钟周期长度}$

$$\text{CPU 时间} = (\text{IC} \times \text{CPI}) / \text{频率}$$

上式表明,CPU 的性能取决于三个要素:①时钟频率;②每条指令所花的时钟周期数;③指令条数 IC。时钟频率取决于硬件技术和组织;CPI 取决于系统结构组织和指令集;指令数目取决于系统结构的指令集和编译技术。

有些时候,在 CPU 的设计中要用到下面一个计算 CPU 时钟周期总数的方法:

$$\text{CPU 的时钟周期数} = \sum_{i=1}^n (\text{CPI}_i \times I_i)$$

其中 I_i 表示 i 指令在程序中执行的次数,CPI _{i} 表示 i 指令所需的平均时钟周期数,这个式子可用来表示 CPU 时间为:

$\text{CPU 时间} = \left(\sum_{i=1}^n \text{CPI}_i \times I_i \right) \times \text{时钟周期长度}$,其中 n 为指令种类数。CPI 表示为:

$$\text{CPI} = \frac{\sum_{i=1}^n (\text{CPI}_i \times I_i)}{\text{IC}} = \sum_{i=1}^n \left(\text{CPI}_i \times \frac{I_i}{\text{IC}} \right)$$

其中, I_i/IC 表示 i 指令在程序中所占的比例。

下面通过例子来说明上述 CPU 性能公式。

例 1.3 如果 FP 操作的比例为 25%,FP 操作的平均 CPI=4.0,其他指令的平均 CPI 为 1.33,FPSQR 操作的比例为 2%,FPSQR 的 CPI 为 20。假设有两种设计方案,分别把 FPSQR 操作的 CPI 和所有 FP 操作的 CPI 减为 2。试利用 CPU 性能公式比较这两种设计方案哪一个更好。

解: 首先我们观察只改变 CPI 而时钟频率和指令条数保持不变的情况。没有采取提高措施之前原系统的 CPI 为:

$$\begin{aligned} \text{CPI}_{\text{原系统}} &= \sum_{i=1}^n \text{CPI}_i \times \left[\frac{I_i}{\text{IC}} \right] \\ &= (4 \times 25\%) + (1.33 \times 75\%) \\ &= 2.0 \end{aligned}$$

采用方案 1(使 FPSQR 操作的 CPI 为 2)后,整个系统的 CPI 为:

$$\begin{aligned} \text{CPI}_{\text{方案1}} &= \text{CPI}_{\text{原系统}} - 2\% \times (\text{CPI}_{\text{老FPSQR}} - \text{CPI}_{\text{新FPSQR}}) \\ &= 2.0 - 2\% \times (20 - 2) \end{aligned}$$

$$= 1.64$$

同样方法可以计算出采用方案 2(提高所有 FP 指令处理速度的措施)后,整个系统的 CPI 为:

$$\begin{aligned} \text{CPI}_{\text{方案2}} &= \text{CPI}_{\text{原系统}} - 25\% \times (\text{CPI}_{\text{老FP}} - \text{CPI}_{\text{新FP}}) \\ &= 2.0 - 25\% \times (4 - 2) \\ &= 1.5 \end{aligned}$$

我们也可以根据以下公式计算出 $\text{CPI}_{\text{方案2}}$

$$\begin{aligned} \text{CPI}_{\text{方案2}} &= (75\% \times 1.33) + (25\% \times 2.0) \\ &= 1.5 \end{aligned}$$

显然,提高所有 FP 指令处理速度的方案要比提高 FPSQR 处理速度的方案要好。

方案 2 的加速比为:

$$\begin{aligned} \text{加速比}_{\text{方案2}} &= \frac{\text{CPU 时间}_{\text{原系统}}}{\text{CPU 时间}_{\text{方案2}}} \\ &= \frac{\text{IC} \times \text{时钟周期} \times \text{CPI}_{\text{原系统}}}{\text{IC} \times \text{时钟周期} \times \text{CPI}_{\text{方案2}}} \\ &= \frac{\text{CPI}_{\text{原系统}}}{\text{CPI}_{\text{方案2}}} \\ &= \frac{2}{1.5} \\ &= 1.33 \end{aligned}$$

例 1.4 假设有两台机器,它们对条件转移指令的处理采用不同的方法,CPU_A 采用一条比较指令来设置相应的条件码,由紧随其后的一条转移指令对此条件码进行测试,以确定是否进行转移。显然实现一次条件转移要执行比较和测试两条指令。CPU_B 采用比较功能和判别是否实现转移功能合在一条指令的方法,这样实现一条条件转移只需一条指令就可以完成。假设这两台机器的指令系统中,执行条件转移指令需 2 个时钟周期,而其他指令只需 1 个时钟周期。又假设在 CPU_A 中,条件转移指令占总执行指令条数的 20%。由于每条转移指令都需要一条比较指令,所以比较指令也将占 20%。由于 CPU_B 在转移指令中包含了比较功能,因此它的时钟周期就比 CPU_A 要慢 25%。现在要问,采用不同转移指令方案的 CPU_A 和 CPU_B,哪个工作速度会更快些?

解: 由上述假设可计算出 $\text{CPI}_A = 0.2 \times 2 + 0.8 \times 1 = 1.2$, $T_{\text{CPUA}} = \text{IC}_A \times 1.2 \times \text{时钟周期长度}_A$ 。CPU_B 由于没有比较指令,使转移指令由原来占 20% 上升为 $20\% \div 80\% = 25\%$,它需 2 个时钟周期,而其余的 75% 指令只需 1 个时钟周期,所以 $\text{CPI}_B = 0.25 \times 2 + 0.75 \times 1 = 1.25$ 。由于 CPU_B 中没有比较指令,因此 $\text{IC}_B = 0.8 \times \text{IC}_A$ 。另外,时钟周期长度_B = 1.25 时钟周期长度_A,所以, $T_{\text{CPUB}} = \text{IC}_B \times \text{CPI}_B \times \text{时钟周期长度}_B = 0.8 \text{IC}_A \times 1.25 \times 1.25 \text{时钟周期长度}_A = 1.25 \text{IC}_A \times \text{时钟周期长度}_A$ 。可见 T_{CPUA} 比 T_{CPUB} 小,所以 CPU_A 比 CPU_B 运行得更快些。

例 1.5 在例 1.4 中,如果 CPU_B 的时钟周期只比 CPU_A 的慢 10%,那么哪一个 CPU 会工作得更快些?

解: $T_{\text{CPUA}} = 1.2 \text{IC}_A \times \text{时钟周期长度}_A$

因时钟周期长度_B = 1.10 时钟周期长度_A, 所以

$$\begin{aligned} T_{\text{CPUB}} &= 0.8 \text{ IC}_A \times 1.25 \times 1.10 \text{ 时钟周期长度}_A \\ &= 1.10 \text{ IC}_A \times \text{时钟周期长度}_A \end{aligned}$$

由于 CPU_B 所需时间较少, 所以 CPU_B 比 CPU_A 运行得更快些。

4. 访问的局部性原理

程序往往重复使用它刚刚使用过的数据和指令。实验表明, 一个程序用 90% 的执行时间去执行仅占 10% 的程序代码。局部性的实质是: 根据程序的最近情况, 可以较精确地预测出最近的将来将要用到哪些指令和数据。局部性分时间上的局部性和空间上的局部性两种。时间上的局部性是指最近访问过的代码是不久将被访问的代码。空间上的局部性是指那些地址上相邻近的代码可能会被一起访问。存储器体系的构成就是以访问的局部性原理为基础的。

1.2.2 计算机系统设计者的主要任务

计算机系统设计者的任务包括指令系统的设计、功能的组织、逻辑设计和其物理实现。它涉及集成电路设计、封装、电源、冷却等问题。要设计出一种最优的方案, 需要熟悉从编译系统和操作系统到逻辑设计和组装等一系列技术。

下面我们列出计算机系统设计者的主要任务。

1. 确定用户对计算机系统的功能、价格和性能的要求

计算机系统设计者必须设计出具有一定功能, 而且价格和性能也令用户满意的系统。功能要求是根据市场的需要而定的。应用软件常常对功能的确定起支配作用。如果一批应用软件基于某一种指令集, 那么设计者必须新的系统结构上实现这个指令集。如果某一类专门的应用有很大的市场, 那么设计者必须考虑新的系统能适应这种应用。

具体的功能要求包括:

(1) **应用领域** 是专用还是通用? 是面向科学计算还是面向商用处理? 专用机应用于特殊领域, 要求有非常高的性能。通用机对各种应用都适合, 有比较平衡的性能。科学计算应用领域要求浮点运算性能高。商用系统要求支持 COBOL、数据库和事务处理。

(2) **软件兼容层次** 如果要求在程序设计语言层兼容, 则对设计者来说是最灵活的, 但需要新的编译器。如果要求在目标代码或二进制代码层兼容, 则系统结构完全确定了, 灵活性差, 但不需要在软件或程序移植方面投资了。

(3) **操作系统需求** 如地址空间大小, 这是很重要的特性, 会限制应用范围。如存储管理, 现代操作系统要求有这一功能, 可采取分页或分段管理方式。如保护, 不同的操作系统和应用要求页保护或段保护。

(4) **标准** 如浮点数标准, 已存在 IEEE、DEC、IBM 等格式和算法; I/O 总线标准有 VME、Sbus、PCI、SCSI 等; 如网络标准, 要求支持不同的网络, 如 Ethernet、ATM 等; 程序设计语言标准有 ANSI C、FORTRAN 77、ANSI COBOL 等。

2. 软硬件的平衡

一旦所设计机器的功能要求确定下来后, 设计者下一步必须考虑如何优化这一设计。最优设计方案的选择依赖于衡量标准的选择。最通用的标准包括价格和性能, 在给定应用

范围,我们可以通过一系列可以代表这一领域的程序来量化计算机的性能。有些性能在某些应用领域很重要,如可靠性和容错功能在事务处理中非常关键。

优化设计必须考虑硬件和软件的合理分配。一种功能由软件实现还是由硬件实现具有不同的优点。软件实现的主要优点是设计容易、改进简单。硬件实现的好处是速度快,有较好的性能。但硬件实现也并不总是比软件实现快。例如一个先进的算法用软件实现其性能优于用硬件实现的一个较差的算法。软件与硬件的平衡才能得到性能价格比最佳的计算机。

有时,一种特殊的要求需要配置相应的硬件,如一台需进行大量浮点运算的用于科学计算的计算机几乎肯定要配置处理浮点操作的硬件,这不是功能问题,而是性能的问题。浮点操作也可以采用软件实现,但速度太慢而导致失去竞争力。而一台使用 COBOL 语言处理商业应用的计算机要进行大量的十进制数和字符串操作,因此,许多系统结构包含具有这些功能的指令。而另外一些计算机则通过软件和标准的整数和逻辑操作来提供这些功能。这是协调软硬件实现的典型例子。

在两种设计方案的选择过程中,设计的复杂性是必须考虑的一个因素。复杂设计需要较长的完成时间。这意味着这样的设计需要有更高的性能才具有竞争力。一般来说,用软件实现复杂设计比用硬件容易一些。因此,可以将某些功能由硬件实现改为用软件实现。另一方面,指令系统和组织结构的设计选择会影响实现的复杂性以及编译器和操作系统的复杂性,设计者必须考虑他所选择的设计方案在硬软件两方面实现的难易程度。

3. 设计出符合今后发展方向的系统结构

一个成功的系统结构应该能经得住软、硬件技术的发展和变化的应用。因此设计者必须特别注意计算机应用和计算机技术的发展趋势,这样才能延长一种机器的使用寿命。

软件的发展趋势也应受到注意。软件发展最重要的趋势之一是程序及其数据占用越来越多的存储容量,程序所需存储容量平均每年增长 1.5 到 2 倍。因此相应地要求地址每年能增长 0.5 位到 1 位。过去 20 年里,另一个重要趋势是用高级语言代替汇编语言。这就加重了编译器的负担,也对系统结构对编译器的支持提出了要求。设计者必须对编译技术有所了解,因为它成为用户和机器之间的最基本的界面。还有一个重要趋势是编程模式的变化也需要系统结构的支持。但新的编程模式的出现速度比编译技术的发展慢得多,大约 10 年时间程序语言才能出现一次出色的改进。

1.2.3 计算机系统设计的主要方法

计算机系统由多级层次组成。从哪一层开始设计就构成了“由下往上”、“由上往下”和“由中间开始”三种设计方法。

1. “由下往上”(bottom-up)设计

它根据硬件技术条件,特别是器件水平,首先把微程序机器级和传统机器研制出来。在此基础上,再设计操作系统、汇编语言、高级语言等虚拟机器级。最后设计面向应用的虚拟机器级。这种设计方法在计算机早期设计中广为采用,其原因是那时硬件成本昂贵,硬件技术水平较低等,计算机设计中更关注硬件结构,而软件技术往往处于被动地位。这种方法容易使软件和硬件脱节,整个计算机系统的效率降低。在硬件技术飞速发展,而软件

技术发展相对缓慢的情况下,这种由下往上的设计方法很难适应系统的设计要求,因而也就很少被采用了。

2. “由上往下”(top-down)设计

这是一种自然直观的设计方法。首先确定用户级虚拟机器的基本特征、数据类型和基本命令等,而后再逐级向下设计,直到由硬件执行或解释那级为止。当然每级设计过程中,既要考虑实现的方法,也要考虑如何使上一级能优化实现。

3. “由中间开始”(middle-out)设计

这里“中间”是指多级层次结构的某两级的界面。多数计算机设计时把“中间”取在传统机器级与操作系统机器级之间。首先对这个界面进行详尽的功能描述与软、硬件功能分配。再由中间点往上、往下同时进行设计。软件系统从操作系统、汇编、编译系统设计,硬件从传统机器级、微程序机器级、数字逻辑级进行设计。软件设计与硬件设计同时进行是中间开始设计的一个优点。

1.3 系统结构的评价标准

评价一个计算机系统结构好坏的标准是什么?我们可以用速度、程序和数据的容量、功耗、体积、编程的难易程度、成本等指标来评价。其中最重要的是性能和成本这两个指标。

1.3.1 性能

1. 主要标准

时间是衡量计算机性能的标准。同样的工作量,花费的时间越少速度就越快。程序的执行时间以秒数记,所以越少的时间代表着越好的性能。

对于时间,我们常常指的是响应时间,也就是完成一个任务的全部时间,包括磁盘访问时间、存储器访问时间、I/O 访问时间等等。然而,在多道程序时,CPU 在一个程序处于 I/O 等待时会转去执行另外一个程序,所以这样就会减少响应时间。

CPU 时间指的是 CPU 计算的时间,它不包括 I/O 等待时间,它还分为用户 CPU 时间和系统 CPU 时间。前者是 CPU 花费在用户程序上的时间,后者为花费在操作系统上的时间,CPU 的性能指的就是用户 CPU 时间。

衡量机器性能的唯一固定而且可靠的标准就是真正执行程序的时间,而其它各种替代标准可能导致错误的结论,而使设计失败。下面介绍几种较流行的替代标准及其不足。

(1) MIPS (million instructions per second)

它表示每秒百万条指令数。对于一个给定的程序,MIPS 定义为:

$$\text{MIPS} = \frac{\text{指令条数}}{\text{执行时间} \times 10^6} = \frac{\text{时钟频率}}{\text{CPI} \times 10^6}$$

程序的执行时间为:

$$T_e = \frac{\text{指令条数}}{\text{MIPS} \times 10^6}$$

既然 MIPS 是单位时间内的执行次数,所以机器愈快其 MIPS 愈高,这一点是比较容易理解的,尤其是对于用户来说。

但是 MIPS 有三个方面的缺陷:

1) MIPS 依赖于指令集,所以用 MIPS 来比较指令集不同的机器的性能好坏是很不准确的。

2) 在同一台机器上,MIPS 因程序不同而变化,有时是很大的。

3) MIPS 可能与性能相反!

最后一种情况的典型例子就是具有可选硬件浮点运算部件的机器。因为浮点运算远慢于整数运算,所以很多机器提供了可选的硬件浮点运算部件,但是软件实现浮点运算的 MIPS 高,然而硬件实现浮点运算的时间少,这时 MIPS 与机器性能恰好相反。类似的情况在具有优化功能的编译器中也可发现。

(2) MFLOPS(million floating point operations per second)

另一种替代标准是 MFLOPS,即每秒百万次浮点操作次数。

$$\text{MFLOPS} = \frac{\text{程序中的浮点操作次数}}{\text{执行时间} \times 10^6}$$

显然,MFLOPS 取决于机器和程序两个方面。所以 MFLOPS 只能用来衡量机器浮点操作的性能,而不能体现机器的整体性能。例如编译程序,不管机器的性能有多好;它的 MFLOPS 不会太高。

然而,因为 MFLOPS 是基于操作而非指令的,所以它可以用来比较两种不同的机器。因为同一程序在不同的机器上执行的指令可能不同,但是执行的浮点运算却是完全相同的。然而 MFLOPS 也并非可靠,因为不同机器上浮点运算集不同,例如 CRAY-2 没有除法指令,而 Motorola 68882 却有。另外 MFLOPS 还依赖于操作类型。例如 100% 的浮点加要远快于 100% 的浮点除。单个程序的 MFLOPS 值并不能反映机器的性能。所以 MFLOPS 也不是一个十分有用的替代标准。

(3) 用基准测试程序来测试评价机器的性能

只有每天都在使用计算机的用户才是评论机器性能的最佳人选。为了评价一个新系统的性能,他只要在该系统上运行由许多程序及操作系统命令组成的任务,然后比较它的执行时间就可以了。然而,很少有人能有条件这样做。他们只能靠别的手段来评价机器的性能。希望这些方法能够给出他们使用的机器的性能参数。在这种情形下有四种级别的程序可用。下面按其评价准确性递减的顺序分别予以列出:

1) 实际的应用程序方法:运行例如 C 编译程序、Tex 正文处理软件、CAD 工具 Spice 等等。

2) 核心程序方法:人们已做出很多的尝试,从实际的程序中抽取少量关键循环程序段,并用它们来评价机器的性能。Livermore Loops 和 Linpack 就是最好的例子。与实际程序不同,任何用户都不会去真正运行这些核心程序。它们的存在只是用来评价性能。

3) 玩具基准测试程序:玩具基准测试程序通常只有 10~100 行而且运行结果是可以预知的。例如 Sieve of Erastosthenes, Puzzle 和 Quicksort 等程序,因为小而且容易键入

并且适用于任何机器而受欢迎。

(4) 综合基准测试程序

综合基准测试程序类似于核心程序,但考虑了各种操作和各种程序的比例,Whetstone 和 Dhrystone 是典型代表。与核心程序相似,没有任何用户真正运行综合基准测试程序,因为它们都不会得出用户能使用的任何东西。与核心程序相比,综合基准测试程序与实际应用的差别更大,因为核心程序是从实际的程序中抽象出来的,而综合基准测试程序是为了体现平均执行而人为编制的。

2. 性能的比较

假如我们能够在选用程序、实验环境和“快”的定义方面取得一致的话,我们就能够对几台机器性能进行比较。然而,结果却往往不是这样的,下面举例说明。

表 1.1 是两个程序在三台计算机上的执行时间。

表 1.1 程序 1 和程序 2 在三台机器上的执行时间

	A 机	B 机	C 机
程序 1	1 秒	10 秒	20 秒
程序 2	1000 秒	100 秒	20 秒
总时间	1001 秒	110 秒	40 秒

我们会得出:

A 机执行程序 1 的速度是 B 机的 10 倍;

B 机执行程序 2 的速度是 A 机的 10 倍;

A 机执行程序 1 的速度是 C 机的 20 倍;

C 机执行程序 2 的速度是 A 机的 50 倍;

B 机执行程序 1 的速度是 C 机的 2 倍;

C 机执行程序 2 的速度是 B 机的 5 倍。

如果孤立地看,上面任何一条都正确,然而综合起来考虑,A、B、C 三种机器的性能谁好谁坏就很难说清楚了。那么怎样进行比较呢?

(1) 总执行时间:一致的衡量标准

评价相关性能的最简单的办法是用这两个程序的总执行时间,那么就会有:

B 机执行程序 1 和程序 2 的速度是 A 机的 9.1 倍;

C 机执行程序 1 和程序 2 的速度是 A 机的 25 倍;

C 机执行程序 1 和程序 2 的速度是 B 机的 2.75 倍。

这种评价的根据是执行时间,它是我们的最终评价标准。

平均执行时间是各执行时间的算术平均值

$$A_m = \frac{1}{n} \sum_{i=1}^n T_i$$

其中 T_i 是第 i 个程序的执行时间。

如果性能是用速度(例如 MFLOPS)表示,那么平均时间是调和平均:

$$H_m = \frac{n}{\sum_{i=1}^n \frac{1}{R_i}}$$

其中 $R_i = 1/T_i$ 。

(2) 加权执行时间

怎样的程序混合比例是最合适呢？程序 1 和 2 在一个任务中的比重相等吗？如果不是，那么会有两种方法来评价性能。第一种方法是给每个程序一个比例权因子 W_i 。比如某个任务中程序 1 占 20% 而程序 2 占 80%。那么它们的权分别为 0.2 和 0.8。将权因子和执行时间的积相加，这叫做加权算术平均值：

$$A_m = \sum_{i=1}^n W_i \cdot T_i$$

其中的 W_i 为第 i 个程序在任务中所占的比重，而 T_i 是该程序的执行时间。表 1.2 列出表 1.1 中数据的加权值。

表 1.2 加权执行时间

	A	B	C	W(1)	W(2)	W(3)
程序 1	1.00	10.00	20.00	0.50	0.909	0.999
程序 2	1 000.00	10.00	20.00	0.50	0.091	0.001
加权算术平均 W(1)	500.50	55.00	20.00			
加权算术平均 W(2)	91.82	18.18	20.00			
加权算术平均 W(3)	2.00	10.09	20.00			

加权调和平均的定义为 $\left(\sum_{i=1}^n \frac{W_i}{R_i}\right)^{-1}$ 。它所体现的性能和加权算术平均相同。

第二种方法是将一个任务的执行时间标准化为一个参考机器的执行时间。

参考机器的执行时间称为平均标准化时间，可以用算术平均和几何平均来表示。几何平均公式为：

$$G = \sqrt[n]{\sum_{i=1}^n \text{ETR}_i}$$

ETR (execution time ratio)_i 是程序标准化为参考机器后的时间，几何平均具有如下性质： $\frac{G(X_i)}{G(Y_i)} = G\left(\frac{X_i}{Y_i}\right)$ 。与算术平均相比，由于上述性质，几何平均在反映性能上可以与机器无关，所以算术平均不可用来求平均标准时间。

因为加权算术平均中的权是在一台给定机器上按比例给定的，所以它不仅受在任务中使用频度的影响，而且还要受具体机器及输入量的制约，而标准化执行时间的几何平均和程序的执行时间无关，而且与使用的机器也无关。

理想的解决方法是根据一个真实任务中程序的执行频度来确定它们的权。假如这点做不到，就进行标准化。另一个问题是输入的非确定性，这个问题的最佳解决方法就是在

比较性能时将输入确定化。如果结果也必须向一个特殊机器标准化,那么先根据正确的权值去估价性能然后再标准化。

1.3.2 成本

1. 成本指标

由于许多人对成本是什么含意还不大清楚,所以有必要作一点解释。对用户来说,计算机系统的成本是指购买系统所要付的钱,即价格。对设计者来说,成本的定义就没有那样清楚了。大多数情况下,成本是指生产成本,其中包括开发工具的折旧费用。

在计算机早期年代,软件是随出售的硬件免费赠送的,但随着计算机工业的不断发展,软件本身变成了很有价值的商品。昔日一度作为免费赠送的软件现已成为在计算机系统的预算中占很大部分的产品。软件和硬件的成本变化趋势如图 1.5 所示。从图中我们可以看到,软件的成本随着其复杂性和长度的增加而不断提高,并没有因为软件工具的不断改进而得到明显的缓解。图 1.5 中黑方块表示在同一时期硬件成本的大致趋势。硬件的成本以不可思议的速度下降。如果软件和硬件的成本还是以这种趋势发展下去,那么 10 年到 20 年后,硬件就可能成为在其上运行的软件的附属品而免费赠送了。当然,这种观点现在看来是相当可笑的。

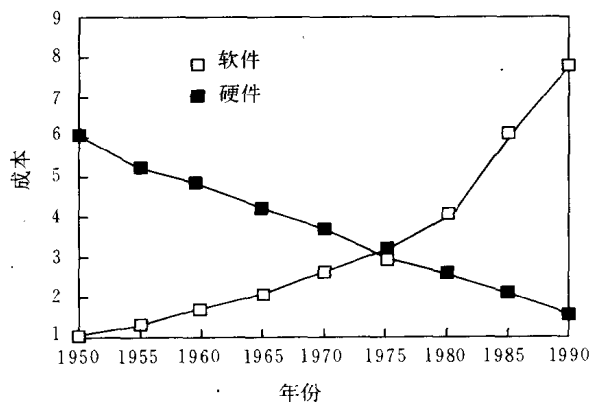


图 1.5 硬件成本和软件成本的变化趋势

软件和硬件的成本各由两部分组成:

(1) 一次性开发成本。

(2) 每个部件的生产成本。不管是硬件还是软件,每种产品的实际成本是该产品产量的函数,如图 1.6 所示。图中曲线是全部产品的累加成本。我们从图中可以看到第一个产品的成本等于开发成本。成本曲线随着产量的增加不断上升。但产量不断增加时,成本曲线上升的趋势变慢,其原因是随着产量的增加,生产经验不断丰富,每个产品的生产成本下降了。产品的单价等于图中曲线值除以产品的数量再加上利润值。如果某个产品的开发成本很高,那么产品的单价与产品的产量就很有关系了。

在软件基本上是免费提供的年代,软件的开发成本或者包括在硬件的开发成本里,其

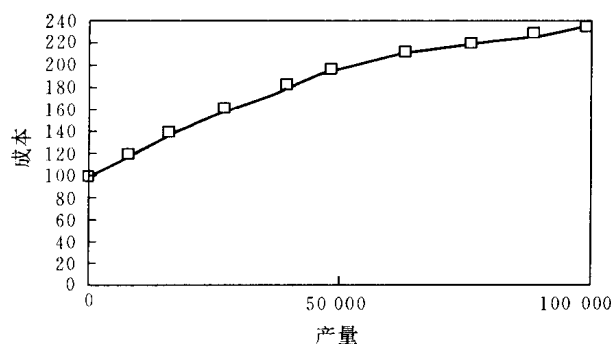


图 1.6 成本与产量的关系

费用由准备自己开发大量软件的用户承担,或者干脆不考虑,或者由软件生产者承担。由于当时硬件的生产成本非常高,软件的生产成本同硬件比起来就低多了。只要软件的开发成本不必通过直接卖软件来补偿,那么免费赠送软件还是合理的。

渐渐地,软件的开发成本越来越高,再也不容忽视,但是复制一个软件基本上不需要费用。这一点和大规模生产 VLSI 芯片类似。这种芯片的开发成本和在其上运行的操作系统的开发成本或和数据库管理应用程序包的开发成本差不多。

在生产数量很大的情况下,每个芯片的生产成本与软件的生产成本差不多。然而芯片以软件的十分之一价格出售,而包括这种芯片的计算机系统的价格却是软件成本的十倍。从这一点来看,芯片、软件和计算机系统的价格似乎和生产成本无关。价格和销售量有关,其原因是价格必须包括开发成本。同一种芯片的销售量可达几百万片,而数据库管理软件的销售量只有几十万份,仅此一项就可以造成十倍的价格差距。

下面我们分析在今后几年中与软件成本相比,硬件成本仍将很高的原因。这里的成本是指生产成本。如今,软件的生产成本很低,并且可以进一步下降,所以在市场竞争中,软件的价格主要是软件的开发成本。

虽然每个芯片的生产成本已经很低了,但整个硬件的生产成本仍然要比软件的生产成本高许多倍。如今复制一个软件的花费要比复制一个硬件的花费少得多。硬件需要装配和测试以保证每一复制品确实满足原始设计的要求。软件复制只需同源程序进行比较,看看其中每一位的内容是否相同就可以确定其准确度,因此硬件复制比软件复制显然困难多了。此外,硬件成本既包括每个部件的生产成本还包括开发成本,而软件成本基本上是开发成本。当计算机只由一个芯片构成时,那么这种计算机的成本和软件成本有点类似。再生产一台由几百或几千个部件组成的计算机要比复制一个软件包复杂得多。最起码,硬件生产者要对芯片和系统进行测试,而软件的复制过程非常简单。因为软件复制的费用低而且可靠,检验起来也不花什么钱,所以在竞争激烈的市场中,随着时间推移,中或高性能计算机把那些配套的软件送给用户的情况已是不太可能出现了。

2. 硬件考虑

另一种错误说法是将来进行系统结构设计时可以毫不吝惜地使用硬件。计算机系统结构设计者认为按目前硬件价格大幅度下降的势头,硬件的成本将非常低,所以我们能够

用比今天复杂得多的硬件来构造系统。诚然,如果成本相同,那么今后的系统会比目前的系统功能更强也更复杂,但这一结论并不意味着能随便浪费硬件。

假设系统 A 的逻辑结构复杂度是目前某个系统的 100 倍,系统 B 的性能基本上和系统 A 相同,而它的逻辑结构复杂度只是目前这个系统的 10 倍或 20 倍,那么系统 A 在竞争中将处于非常不利的地位。若系统 A 能多卖出几百台或几千台,那么系统 A 的价格还可能和系统 B 竞争。如果系统 A 和系统 B 的销售量差不多,那么系统 A 的低效率使得它的价格要比系统 B 高,当然这里假定两个系统都是由相同技术的部件组成的。如果系统 A 的芯片的集成度是系统 B 的芯片的集成度的 10 倍,并且系统 A 的每个芯片的价格仅是系统 B 的每个芯片价格的 $1/10$,那么上述结论就会发生变化,即是器件技术而不是系统结构技术决定系统的价格了。

本书还将研究用低成本部件构成系统结构的方法,同时需要注意这些系统结构的效率问题,以确保我们能采用这些低成本部件。

例如,有一个没有共享存储器的多处理机系统,若要在其上面运行一个并程序,方法之一是把这个程序装入所有处理机。当这个程序较小或者处理机数目不太多时,这个程序的多个副本所占用的存储器容量还是可以接受的。但是,如果程序长达 1 兆字节,且系统有 1 000 台处理机,那么这个程序占用的存储空间将达 1 000 兆字节。如果有办法使所有处理机能共享一个程序副本,那么就不需要这么大的存储容量了。

如果系统 A 要多个程序副本,而系统 B 由于设计巧妙,达到同样的性能却只要一个程序副本,那么系统 A 所要求的 1 000 兆字节的存储器将使它无法和系统 B 竞争,除非存储器的价格很便宜,以致 1 000 兆字节的存储器的成本在整个系统中只占很小的部分。系统 A 的设计者总希望存储器的价格不久能下跌,但是系统 A 需要增加 10^{10} 位存储器,而这是一个巨大的数字。如果按目前价格下跌的趋势继续下去,每位存储器的价格下跌到能使 10^{10} 位存储器的总价格也很低,那大约需要 20 至 30 年。

显然,系统 A 的设计者必须寻找其他方法以便克服系统 A 的这个缺点,使它适于某些特殊应用。如果系统 A 的每台处理机执行不同的程序,那么在这种应用场合系统 A 的总效率将很高。而系统 B 在同样的应用场合将很难与系统 A 相比。

系统结构设计者应把自己的系统在多个不同的应用场合使用,搞清楚使用情况,用这种方法来量测系统的性能。由于应用场合不同,系统的效率可能变化很大。上述这种测量系统性能的方法能揭示某种系统结构在哪些应用场合能使用户受益,而另一些应用场合采用另一些系统结构会更好一些。

假设一个系统由很大数量的处理机组成,比如有 1 6000 台处理机。还假设当系统解决某个应用问题时, N 台处理机的加速比与 $\log_2 N$ 成比例。那么,1 6000 台处理机的加速比是 $14x$,其中 x 是某个常数。如果这 1 6000 台处理机非常便宜,以致可以忽略它们的成本,而这个应用问题在这一系统上运行要比在单处理机上运行快 $14x$ 倍,增加处理机台数所增加的成本不算大,那么虽然加速比不算高但还是值得的。

假设有一个系统 B,它除了处理机台数只有 128 外,其他和系统 A 相同,它的加速比是 $7x$,系统 A 比系统 B 要复杂 100 多倍,而系统 A 的速度只有系统 B 的 2 倍。这只有当硬件的成本几乎为零时,系统 A 才能和系统 B 竞争。而至少在今后 10 年里,通过用 100

倍硬件来获得 2 倍的性能的方法看来是不尽合理的。

这一节的讨论告诉我们：

- (1) 我们可以用价格和性能这两项指标来评价系统结构的好坏。
- (2) 测量系统结构的效率时必须考虑负载。
- (3) 使用大量器件但效率很低的系统结构是无法与设计简单而效率很高的系统结构竞争的。

1.4 计算机系统结构的发展

1.4.1 冯·诺依曼结构

冯·诺依曼等人于 1946 年提出了一个完整的现代计算机雏型,它由运算器、控制器、存储器和输入输出设备组成,如图 1.7 所示。

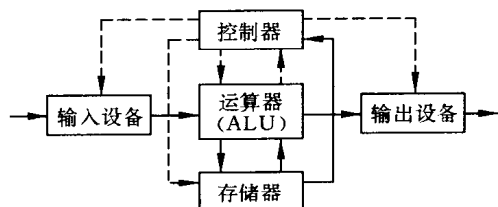


图 1.7 早期的冯·诺依曼型机器组成框图

现代的计算机系统结构与冯·诺依曼等人当时提出的计算机系统结构相比虽已发生了重大变化,但就其结构原理来说占有主流地位的仍是以存储程序原理为基础的冯·诺依曼型计算机。存储程序原理的基本点是指令驱动,即程序由指令组成,并和数据一起存放在计算机存储器中,机器一经启动,就能按照程序指定的逻辑顺序把指令从存储器中读出来逐条执行,自动完成由程序所描述的处理工作。冯·诺依曼计算机的特征可概括为:

- (1) 存储器是字长固定的、顺序线性编址的一维结构。
- (2) 存储器提供可按地址访问的一级地址空间,每个地址是唯一定义的。
- (3) 由指令形式的低级机器语言驱动。
- (4) 指令的执行是顺序的,即一般按照指令在存储器中存放的顺序执行,程序分支由转移指令实现。
- (5) 机器以运算器为中心,输入输出设备与存储器之间的数据传送都途经运算器。运算器、存储器、输入输出设备的操作以及它们之间的联系都由控制器集中控制。

虽然至今绝大多数计算机仍基于上述结构特点,但这四十多年来计算机系统结构有了许多改进。主要包括以下几个方面:

- (1) 计算机系统结构从基于串行算法改变为适应并行算法,从而出现了向量计算机、并行计算机、多处理机等。
- (2) 高级语言与机器语言的语义距离缩小,从而出现了面向高级语言机器和直接执

行高级语言机器。

(3) 硬件子系统与操作系统和数据库管理系统软件相适应,从而出现了面向操作系统机器和数据库计算机等。

(4) 计算机系统结构从传统的指令驱动型改变为数据驱动型和需求驱动型,从而出现了数据流机器和归约机。

(5) 为了适应特定应用环境而出现了各种专用计算机,如快速傅里叶变换机器、过程控制计算机等。

(6) 为了获得高可靠性而研制容错计算机。

(7) 计算机系统功能分散化、专业化,从而出现了各种功能分布计算机,这类计算机包含外围处理机、通信处理机等。

(8) 出现了与大规模、超大规模集成电路相适应的计算机系统结构。

(9) 出现了处理非数值化信息的智能计算机,例如自然语言、声音、图形和图象处理等。主要的处理方法已不是依靠精确的算法进行数值计算而是依靠有关的知识进行逻辑推理,特别是利用经验性知识对不完全确定的事实进行非精确性推理。

1.4.2 软件对系统结构的影响

由于软件相对硬件的成本越来越贵,软件产量和可靠性的提高越来越困难,出现了“软件危机”。但是随着计算机系统的发展和应用范围的扩大,已积累了大量成熟的系统软件和应用软件。因此,用户就希望在新的计算机系统出台后,原先已开发的软件仍能继续在升档换代的新型号机器上使用,这就要求软件具有可兼容性,即可移植性。它是指一个软件可不经修改或只需少量修改便可由一台机器移植到另一台机器上去运行,即同一软件可应用于不同的环境。那么,如何解决软件的可移植性问题呢?根据不同的要求可采用如下三种方法:

1. 采用系列机方法

所谓系列机是指在一个厂家内生产的具有相同的系统结构,但具有不同组成和实现的一系列不同型号的机器。如IBM370系列机有370/115、125、135、145、158、168等一系列从低速到高速的各种型号。它们具有相同的系统结构,而采用不同的组成和实现技术,有不同的性能和价格。它们有相同的指令系统,但在低档机上指令的分析和指令的执行顺序进行,而在高档机上采用重叠、流水和其他并行处理方式。从程序设计者来看,各档机器却具有相同的32位字长,但从低档到高档机器,其数据通道的宽度分别为8位、16位、32位、甚至64位。DEC公司先后推出VAX-11/780、750、730、725、MICRO-VAX、785等型号。它们都具有32位的字长,都运行VAX/VMS操作系统,都具有相同的I/O连接和使用方式。但在组成上,VAX-11/780通过同步底板互连总线SBI把CPU、主存子系统、控制台子系统、单总线适配器和多总线适配器连接起来。但VAX-11/730没有多总线,而VAX-11/750甚至没有SBI,也没有控制台子系统,而集中控制台取而代之。采用哪种总线结构对程序员来讲是透明的。

一种系统结构可以有多种组成。同样,一种组成可以有多种物理实现。系列机从程序设计者看具有相同的机器属性,因此按这个属性编制的机器语言程序以及编译程序都能

通用于各档机器,所以各档机器是软件兼容的,即同一个软件可以不加修改地运行于系统结构相同的各档机器,可获得相同的结果,差别只在于不同的运行时间。系列机较好地解决了软件要求环境稳定和硬件、器件技术迅速发展之间的矛盾,达到了软件兼容的目的。但是,系列机为了保证软件的兼容,要求系统结构不准改变,这无疑又成为妨碍计算机系统结构发展的重要因素。实际上,为适应性能不断提高或应用领域不断扩大的需要,系列机后面出来的各档机的系统结构应允许有所发展和变化。但是,这种改变只应该是为提高机器总的性能所作的必要扩充,而且往往只是改进系统软件的性能出发来修改系统软件(如编译系统),尽可能不影响高级语言应用软件的兼容,尤其是不允许缩小或删除运行已有软件的那部分指令和结构。例如,在后出的各档机器上,可以为提高编译效率和运算速度增加浮点运算指令,为满足事务处理增加事务处理指令及其所需功能,为提高操作系统的效率和质量增加某些操作系统专用指令和硬件等等。因此,系列机的软件兼容分为向上兼容、向下兼容、向前兼容和向后兼容四种。向上(下)兼容是指按某档机器编制的程序,不加修改就能运行于比它高(低)档的机器。向前(后)兼容是指按某个时期投入市场的某种型号机器编制的程序,不加修改就能运行于在它之前(后)投入市场的机器,这样,对系列机的软件向下和向前兼容可以不作要求,向上兼容在某种情况下也可能做不到(如在低档机器上增加了面向事务处理的指令),但向后兼容却是肯定要做到的。

我们把不同公司厂家生产的具有相同系统结构的计算机称为兼容机。它的思想与系列机的思想是一致的。例如,Amdahl 公司照搬 IBM 370 的系统结构,以便充分利用 IBM 370 的软件,却又采用新的组成、实现和器件工艺,研制出在性能价格比方面超过 370 的 Amdahl470、480 等。兼容机还可以对原有的系统结构进行某种扩充,使之具有更强的功能,例如长城 0520 与 IBM PC 兼容,但有较强的汉字处理功能,使之具有较强的市场竞争能力。

2. 采用模拟与仿真方法

系列机方法只能在具有相同系统结构的各种机器之间实现软件移植。为了实现软件在不同系统结构的机器之间的相互移植,就必须做到能在一种机器的系统结构上实现另一种机器的系统结构。从指令系统来看,就是要在一种机器的系统结构上实现另一种机器的指令系统。一般可采用模拟或仿真方法。

模拟方法是指用软件方法在一台现有的计算机上实现另一台计算机的指令系统。如在 A 计算机上要实现 B 计算机的指令系统,通常采用解释方法来完成,即 B 机器的每一条指令用一段 A 机器的指令进行解释执行,如同 A 机器上也有 B 机器的指令系统一样。A 机器称为宿主机,被模拟的 B 机器称为虚拟机。为了使虚拟机的应用软件能在宿主机上运行,除了模拟虚拟机的指令系统外,还需模拟其存储体系、I/O 系统、控制台的操作,以及形成虚拟机的操作系统。由于模拟是采用纯软件解释执行方法,因此运行速度较慢,实时性差,因此模拟方法只适合于移植运行时间短,使用次数少,而且在时间关系上没有约束和限制的软件。

如果宿主机本身是采用微程序控制,则对 B 机器指令系统每条指令的解释执行可直接由 A 机器的一段微程序解释执行。这种用微程序直接解释另一种机器指令系统的方法称为仿真。A 机器称为宿主机,B 机称为目标机。为仿真所编写的解释微程序称为仿真微

程序。除了仿真目标机的指令系统外,还需仿真其存储系统、I/O 系统、控制台的操作等。模拟方法中模拟程序存放在主存中,而仿真方法中仿真微程序存在控存中,因此仿真的运行速度要比模拟方法快。用仿真方法时,由于微程序机器级结构更依赖计算机的系统结构,因此对于系统结构差别较大的机器难于完全用仿真方法来实现软件移植,所以通常将模拟和仿真这两种方法混合使用,对于使用频率较高的指令,尽可能用仿真方法以提高运算速度,而对使用频率低且难于用仿真实现的指令则用模拟方法来实现。

3. 采用统一的高级语言方法

如果能采用一种可以满足各种应用需要的通用高级语言,那么用这种语言编写的应用软件可移植问题就解决了。如果操作系统的全部或一部分是用这种高级语言编写,则系统软件中的这部分也可以移植,所以采用统一高级语言来编写应用软件和系统软件是实现软件移植的一种方法。这种方法可以解决结构相同或完全不同的各种机器上的软件移植。要统一高级语言,语言的标准化很重要,但难以在短期内解决。

1. 4. 3 价格对系统结构的影响

我们将介绍各种改进系统性能的方法和技术。要较全面地评价一个系统结构时,既要考虑性能又要考虑价格。目前大多数情况下,大家把性能和价格当作一个参数看待,即性能/价格比。如果你把一台机器的速度提高了 10%,那么用户一般是愿意多付 10%的钱,即这台机器的性能/价格比保持不变。如果价格提高 10%而机器的性能是提高 20%,那么用户会更愿意购买这台新机器,可以得到真正的实惠,实际上这台机器有较高的性能/价格比。如果价格提高 10%而性能只提高 5%,即单位计算的成本增加而不是减少,那么大多数用户对这台机器就不会感兴趣。当然也有例外的情况,例如已有的某台机器的解题能力已不能满足用户的需要,用户迫切需要一台容量更大速度更快的机器,这时的性能/价格比另有解释了。因为用户有了较大解题能力的机器所获得的实惠要比所付出的钱合算。实际情况是用户为了得到大的解题能力,单位计算的成本增加了,所带来的好处是过去不能做的计算现在能在这台机器上完成了。然而,用户在购买这种解题能力很强的机器时如果有几种方案可供选择,虽然所有方案的性能/价格比都比用户现有的系统要低,但可以选择一个性能/价格比最高的系统。

前面列举的例子都是性能和价格相差不大的系统,这种情况下用性能/价格比这个参数能较好地反映系统之间的相对性能。如果两个系统的性能和价格相差很大,那么性能/价格比这个参数就没有实用意义了。

例如,用性能/价格比这个参数对一台八位电子游戏机和一台功能很强的 CAD 工作站进行比较时,往往会得出使人误解的结论。这两个系统都能显示图象和实时交互式地处理图象。如果我们有办法测出它们的相对性能,那么电子游戏机的性能/价格比会比工作站好得多。问题是两个系统的价格之比高达 1000 : 1,虽然两者的性能之比也很大,但不会大到 1000 : 1。这个例子告诉我们为了使性能/价格比这个参数能说明问题,进行比较的两个系统的功能须比较类似或性能相对接近。

上述讨论指出了改进系统结构的两条重要途径:

(1) 性能或价格较小的变化产生比原系统好的性能/价格比。这个“小”是指性能或价

格变化 10 倍左右的情况。

(2) 提高系统的绝对性能,而价格增加比较合理。这种方法实际上会使性能/价格比下降,但只要用户有能力支付这笔增加的费用,并认为支付的费用所获得的高性能给他们带来的实惠是值得就行了。

由于通货膨胀等原因,用货币表示成本有时不太说明问题。因此用能反映成本高低的其他参数来定义成本可能更合理。例如用插头数、芯片面积、芯片引脚数、功耗等物理参数;设计时间、软件大小、开发队伍人数等和研制有关的参数。

本书无法列出影响成本的全部因素,只能讨论最重要的几个因素。在两个只有几个关键设计不同而其他都很相近的系统结构进行比较时,把影响成本的最主要部分孤立出来是很有必要的。这样可以把注意力放到两者的差别上,集中研究它们对成本的影响。每种方案都有它的优点和缺点,优点和缺点反过来又影响该方案的成本。有时我们很难给出某种方案的绝对费用是多少,但可以给出各种设计对成本影响的大小。我们可以用对成本影响最大的因素去评价一个系统。

1.4.4 应用对系统结构的影响

由于技术的不断进步,解决高性能计算机系统结构设计中出现的问题有了新的方法。例如,新技术使得高度并行成为可能,增加处理机数目可以获得高性能等。那么提高单处理机性能的研究还需要继续进行吗?我们认为不能简单地说要或不要,而是需要把各种研究方法结合起来。有一点可以肯定,我们所选择的方法几乎都与具体应用问题有关。

应用对高性能系统结构的设计影响很大,因为想从系统结构中获得最大可能的解题能力需要非常高的成本。一个系统的效率高低是关键问题,低效率是最大的浪费。如果某个应用领域的负载是确定的,那么可以设计出面向这种负载的系统结构。这种系统结构可以重点配备与特殊负载有关的功能部件,而为满足处理一般问题而设置的功能部件就不必在这种系统结构中出现。

我们的目的是要确保系统结构中所有功能部件对获得高性能都作出贡献,这样,系统的效率就提高了。如果能构造一台处理所有问题都有很高效率的通用计算机,那是再好不过的事。我们不能排斥今后会有这种可能性。然而,在今后 10 年里,一些特殊领域要求非常高的计算速度,需要专门为这些领域设计高效率的系统结构。这些重要的领域包括:

- (1) 高结构化的数值计算——气象模型、流体流动、有限元分析;
- (2) 非结构化的数值计算——蒙特卡洛模拟、稀疏矩阵;
- (3) 实时多因素问题——语音识别、图象处理、计算机视觉;
- (4) 大存储器容量和输入输出密集的问题——数据库系统、事务处理系统;
- (5) 图形学和设计系统——计算机辅助设计;
- (6) 人工智能——面向知识的系统、推理系统。

显然,数值计算领域要求系统结构中有高精度的浮点处理机,计算要求较高的应用场合可能还需要成千上百台这种处理机。图形系统需要大量的定点计算,以便对窗口和透视图提供支持。浮点运算在某些需要平滑曲线真实显示和射线跟踪计算的图形应用系统中起重要的作用。人工智能系统一般不需要很强的运算能力,但通常需要很大的存储器

容量。

如果一台高性能计算机系统结构要同时满足上述各个领域的要求,那么当它处理某个具体应用时,必然会出现低效率的情况。其原因是对某个应用问题来说,系统结构的大部分功能可能没有用上。专门为某个领域设计一台专用的计算机可能会比通用的计算机更有吸引力,因为这种专用计算机的价格可能会比较低。当然这种专用机的价格还取决于是否有足够大的市场,以便把开发费用分摊到各台机器上去。如果只卖出少量几台,那么它的优势也就没有了。因此,即使是专用的高性能机器也应该尽可能地在它应用领域有一定的通用性。当专用的系统结构扩大了适用范围后,它的市场也就大了,但同时硬件的利用率也降低了。系统结构设计者面对着如何在两者之间进行权衡的问题。设计的基本出发点是使专用的系统结构的高效率与通用系统结构的广泛市场成均势。

系统结构设计者必须在专用系统结构和通用系统结构两者之中找一个具有竞争价格和高性能的新的设计方案。设计的决策会随着时间的推移而变化,其原因是设计决策既和开发费用有关还和每个部件的生产费用有关,而这两项费用随着技术的不断进步变化很大。

1.4.5 VLSI 对系统结构的影响

由于 VLSI 的发展,计算机系统的价格发生了很大的变化。在 20 世纪 50 年代,硬件非常贵,一个用户买不起一台有 128K 存储容量的计算机,所以几个用户共同出钱买一台大的计算机。在一台计算机上几个用户分时地运行他们的程序,从而减少了存储器、处理器、输入输出设备的空闲时间。那时看来使用大型计算机在经济上是合算的。随着计算机解题能力的不断增加,上机的用户数目也不断增加,但机器的价格增加却比较少。

在 20 世纪 80 年代,VLSI 技术使计算机的价格发生了新的变化。我们不再设法把有 128K 字节存储容量的机器的每一个周期都利用上,现在我们随时就能找到一天 24 小时中大部分时间处于空闲的机器。所以,为了处理众多小规模计算任务,用户买一台能足够处理这种作业的机器就行了。许多用户共享一台大型机看来并不非常经济了。现在的情况是小作业在小机器上运行,大作业在大机器上运行。其实今天的“小”机器和 25 年前的“大”机器有差不多的计算能力,而这种“小”机器过去通常 100 个用户共享,而现在是一个用户单独占用了。

今天大型机或由小型机构成的网络支持许多并发的用户,主要是共享数据而不是共享机器周期。我们还能看到许多用户共享一台超级计算机的情况,其原因是这种机器对任何一个用户来说实在太贵了。

1.4.6 技术的发展对价格的影响

图 1.8 是采用不同器件技术的计算机其性能与价格关系的示意图。其中性能用 MIPS 表示,价格是单位 MIPS 的价格。

在 20 世纪 80 年代中期,占统治地位的器件技术是 MOS 技术,其中主要是 NMOS。所有用这种器件构造的计算机,其每单位 MIPS 的价格几乎是个常数,如图中第一个台阶所示。

第二个台阶是另一种技术,估计是 ECL 的双极性技术。所有用这种器件构造的计算机,其每单位 MIPS 的价格也几乎是个常数。

第三个台阶是一种更特殊的技术,专门用于高性能的计算机。由于有冷却的要求、生产困难、芯片的集成度比较低等原因,这种器件技术的成本最高。所有用这种器件构造的计算机,其每单位 MIPS 的价格也几乎是个常数。

虽然图 1.8 中的曲线是近似的,但它告诉我们,器件技术对性能与价格关系的影响程度。器件决定了一台计算机的基本周期的时间。我们可以用地址和数据通路的宽度乘以时钟频率所得的值,即计算机系统最大信息传输率,来粗略地估计一台计算机的处理能力。

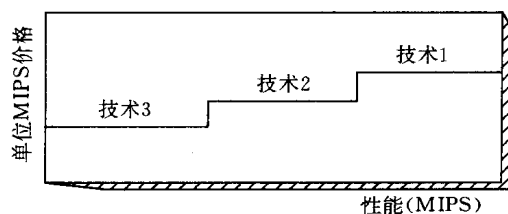


图 1.8 性能、价格与技术的关系

假设采用某种器件,那么设计时大多数高速部件都会采用由这种器件决定的最大或几乎最大的时钟频率。上面提及的计算机处理能力与地址宽度、数据宽度和时钟频率有关。当时钟频率取最大值时,改进性能的方法是把数据通路宽度由 8 位改为 16 位或 32 位,并相应地增加存储器容量。通过这种方法所获得的性能与总线宽度成线性关系,但价格也与总线宽度成线性关系。所以,随着总线宽度的增加,性能提高了,总价格也提高了,但单位 MIPS 的价格仍保持不变。需要指出的是存储器的价格不一定随着总线宽度线性地增加。产生如图 1.8 所示的台阶型曲线是因为假设存储器容量随着性能而线性地增加。即一个速度为 2MIPS 系统的存储器的典型配置可能是 4M 字节,同一个系统如果速度达到 4MIPS 或 8MIPS,则存储器的配置分别为 8M 字节和 16M 字节。

从图 1.8 我们可以得到如下结论:当某一系统结构的性能超过某种器件技术的极限值时,要采用性能更好的器件了。从图中看,曲线移到上一层台阶上去了,这会导致单位 MIPS 价格的提高。

如果性能用 MIPS 来度量,那么我们可以得出:

$MIPS = (\text{指令条数/周期})(\text{周期数/秒}) \cdot 10^{-6}$,其中第一项与系统结构有关,系统结构设计者可以设法提高或降低它。第二项与所采用的器件技术有关。

实际上,第一项(系统结构)和第二项(器件技术)并不像我们上面所说的能绝然分开。例如,第二项的时钟速度就与系统结构中指令译码的复杂性有关。尽管如此,我们还是主要通过研究每个机器周期执行的指令条数这一项来提高系统的性能。下面是三种常用的增加每个机器周期执行指令条数的方法。

(1) 减少要执行的指令条数。通过采用较好的算法,有可能用较少的指令完成同样的工作。

(2) 在系统结构中增加硬件辅助部件,以改进系统结构的效率。例如,增设 Cache 存储器可以增加每个周期执行指令的条数。设置在某些特殊应用场合非常有效的 SORT、SEARCH 指令。

(3) 多条指令并发地执行。利用重复设置的硬件增加每个周期内执行的指令条数。

第一种方法从系统结构角度看可能觉得奇怪。人们可能认为系统结构设计者不必研究算法问题。但事实恰恰相反。因为我们的目的是使系统在某些应用场合既要性能高又要价格低,谁也没有规定解决的方法只许采取系统结构方法。实际上,要降低价格,有时改进算法可能是上述所有方法中最有效的方法。

1.4.7 算法和系统结构

系统结构设计者必须仔细地研究算法,以便了解怎样才能充分利用系统结构的高性能。系统结构设计者还需要了解各种应用问题,例如受浮点除速度限制的问题,受内排序限制的问题,受图象数据位变换的解释能力限制的问题等等。改进原有的算法是最常用的方法。有时候只要作简单的修改,有时候把原有的算法修改成几乎完全是新的算法了。这样,原来很不适于在高性能系统结构上处理的应用问题,就能在只要增加一些便宜硬件的系统结构上处理了。针对某个特殊应用问题,如果算法上有突破,甚至不需要高性能的系统结构,而在普通的系统上就能得到解决。关于浮点除和排序这两类算法已研究得很透彻了,所以无论怎样修改现有的算法,一般不太可能会有新的突破。然而许多其他算法都有改进的余地,所以通过算法的研究会大大提高系统的性能。

虽然我们不能期望计算机系统结构设计者进入应用领域并在这个领域的算法方面产生突破,但是系统结构设计者把一些基本算法修改成更适合于处理的形式是可能的。例如系统结构设计者可以把一个问题按新的方式划分,以便减小存储器工作区和减少所需的高速寄存器的数目。例如系统结构设计者可以找到构造问题的新方法,以便使它适合于并行的系统结构。

系统结构设计者实际上既要研究算法又要研究系统结构。充分利用算法和系统结构两方面优势,构造出一种有效的解决问题的方法。必须提醒的一点是通常应该研究一类算法,而不是一个算法。更困难的是设计一个系统结构,使所有这类问题在此结构上都能很好解决。

第二种解决方法是改进基本的系统结构。过去,我们已经采用了许多不同技术来改进系统的性能。例如,指令缓冲器、Cache 存储器、流水线等,已在许多商业机器中采用。为了减少取指令的次数,有些机器设有复杂的指令。为了减少指令的基本周期时间,有些机器去掉了复杂指令。

系统结构设计者需要了解系统的瓶颈在哪部分,然后采取措施去掉瓶颈。一个设计得很好的系统当它以峰值性能工作时,应该使许多部件都处于接近饱和状态。如果只有一处瓶颈,而其他功能部件都处于没被充分利用的状态,那么我们说这个系统设计得不好。我们可以把这一系统的某些无法被利用的功能去掉,这样整个系统的性能仍保持不变而价格降低了。或者想法解决这个瓶颈,可用较少的费用而达到改进整个系统性能之目的。

第三种方法是利用并行性获得高速度。通常这是最费钱的方法。但是 VLSI 芯片价格

的迅速下降,使并行硬件成为一种可行的方案了。反过来再看图 1.8,它告诉我们,采用低价格的器件技术是获得高性能最理想的方案。但每种器件技术只在某一性能范围才有效。然而并行系统结构提供了一种采用低成本器件技术获得高性能的方法。由于并行计算机可以采用价格较便宜的器件,而高性能串行计算机必须采用价格较贵的器件,这是研究并行系统结构的吸引力之一。当然通过采用低价格器件技术所获得的好处将被由于并行执行程序的开销而抵消。

上述改进性能的三种技术都很重要,但并不是说只有这三种技术。所有技术都值得进一步研究。由于现实世界和本书中所假设的高度理想化的模型不完全一致,所以现实世界可能出现配置不平衡的系统,而根据性能和价格标准很难判断这些配置不平衡系统的好坏。

例如,系统 A 有一个 256K 字节的 Cache 存储器,系统 B 为了能与系统 A 竞争设有 512K 字节或 1M 字节的 Cache 存储器。即使已经证明这么大容量的 Cache 存储器不太合理,但系统 B 已经用较大的 Cache 容量获得了竞争的优势。因此,这些竞争性压力很容易迫使系统结构设计者设计出配置很不平衡的系统。

这几年我们通常用性能和价格来衡量系统的好坏。如果系统在首次推出时,其配置很不平衡,实际的性能不如所说的那样好,用户就会觉得价格太高了。这样,系统的最后配置将被修改成性能和价格都比较合理的情况。

通过上面讨论,我们已经知道为了改进系统的性能,系统结构设计者可以采用改进算法、在基本系统结构上增加硬件辅助部件、设计并行系统结构这三种方法。有时为了达到高性能,可能需要把这三种方法结合起来考虑。

由于新的器件不断推出,系统结构设计者可用的积木块也相应地不断变化,容易实现的新结构也就不断出现了。另外,应用领域也不断变化,许多新领域开始使用计算机,已经使用计算机的领域的应用范围和规模都比原来增大了。总之,系统结构设计者在进行新的设计时会遇到新问题,需要研究新方法、新方案、新结构。但是系统结构作为方法学来说,它是把不同可用的功能部件组合在一起成为一个能高效解决应用问题的系统,这一点是始终不会改变的。

技术的发展促进了计算机系统结构的发展。每年都有新的器件和新的设备问世,使新功能的计算机系统结构不断出现。今天看来富有想象力和卓有成效的系统结构,明天也许会显得落后了。同样,今天看来是荒谬、不切实际的建议,明天可能是很理想的设计方案。因此不存在一个绝对的标准用来断言一种系统结构一定比另一种系统结构好。

学习计算机系统结构的关键是学会在现有技术条件下评价系统结构的方法。了解一台计算机的速度固然重要,但了解其是否充分利用了处理机周期、存储器容量、输入输出带宽等也同样重要。评价一个系统结构既要考虑性能又要考虑价格,不能只注意性能。由于技术的发展,各功能模块的价格每年都有很大的变化,各功能模块之间的相对价格同样也有很大的变化,所以为了获得较好的性能/价格比,不同功能模块的最佳比例也应随着技术的发展而有所改变。

本书将介绍系统结构的设计技术,而不是仅仅给出结论。我们准备提供一系列可供选择的设计技术,其中有些对今天来说是合理的,有些则不合理。我们将介绍如何从中选择

合适的技术来构造高性能的系统和如何评价采用 20 世纪 90 年代技术所制造的系统。评价所获得的结论一直到下个世纪初可能都是合理的,但是我们无论如何也不能说今天看来是最佳的系统结构在下一个十年仍是最佳的。

然而,方法是长期有效的。系统结构设计者使用本书所介绍的设计方法与评价技术就可以在任何时候根据当时的技术,设计出高性能的系统。

性能分析应建立在整个系统结构基础上。设计和分析一个高性能系统非常复杂,最好的方法是把一个大系统分解成由功能模块组成的层次形式,这样每个功能模块的系统结构可以分别进行分析。如果某个功能模块仍然非常复杂,那么还可以进一步分解成更小的功能模块。例如,处理机可以看作由算术部件、控制逻辑、寄存器等组成。

指令系统的设计是系统结构的一个重要内容。目前存在着指令系统是非常复杂好还是非常简单好的争论。我们不打算介入这场争论,因为关于这个争论不可能只有一个答案。但是我们要阐明决定这个答案的各种因素。不管采用什么技术,系统结构设计者在进行新的设计时应应对这些因素进行权衡。

人们往往易把计算机系统结构设计和计算机硬件设计相混淆。由于计算机系统结构的设计是在功能这一层次上考虑问题,当然也不排斥硬件这一层次,但计算机系统结构设计不只包括硬件设计。例如,一台具有算术和逻辑功能的处理器可以由硬件实现而无需额外的程序设计。存储器管理功能可以由硬件和软件共同实现,它们之间的分工取决于性能、价格及当前硬件和软件的可用性情况。在 VLSI 还处于发展初期时,存储器管理功能一般由软件实现,处理机必须提供用于地址变换和地址保护的寄存器。随着 VLSI 的不断发展,存储器管理的大部分功能变为由硬件实现。许多系统已经用硬件来实现过去只能用软件方法实现的存储器管理算法。如果把存储器管理软件写到只读存储器,管理程序可以透明地调用它,那么硬件和软件之间的界线就变得模糊了。我们可把这种只读存储器视为一个实现存储器管理的黑匣子。这样,实现存储器管理功能就由软件方法变为硬件方法了。使用这种只读存储器芯片的系统结构设计者不需要考虑存储器管理软件。如果这种芯片只实现大部分而不是全部的存储器管理功能,那么系统结构设计者还必须通过软件模块来弥补不足的功能。

综上所述,计算机系统结构是把各个功能部件组成一个系统,这些部件可以是硬件、软件或两者的混合体。系统结构设计是选择一种最佳的部件组合,使得整个系统能有效地工作。以后章节我们将介绍不同系统结构的实例,其中有些已被证明是成功的,有些则是可能会成功的方案。

习 题 一

1.1 解释下列术语

层次结构;计算机系统结构;计算机组成;计算机实现;透明性;由上往下设计;由下往上设计;系列机;软件兼容;兼容机;模拟;仿真;虚拟机;宿主机;指令流;数据流;Amdahl 定律;CPI;MIPS;MFLOPS

1.2 如有一个经解释实现的计算机,可以按功能划分成 4 级。每一级为了执行一条指令

需要下一级的 N 条指令解释。若执行第一级的一条指令需 K_{ns} 时间,那么执行第 2、3、4 级的一条指令各需要用多少时间?

- 1.3 操作系统机器级的某些指令就用传统机器级的指令,这些指令可以用微程序直接实现,而不必由操作系统自己来实现。根据你对第 1.2 题的回答,你认为这样做有哪两个好处?
- 1.4 有一个计算机系统可按功能划分成 4 级,各级的指令都不相同,每一级的指令都比其下一级的指令在效能上强 M 倍,即第 i 级的一条指令能完成第 $i-1$ 级的 M 条指令的计算量。现若需第 i 级的 N 条指令解释第 $i+1$ 级的一条指令,而有一段第 1 级的程序需要运行 ks ,问在第 2、3 和 4 级上的一段等效程序各需要运行多长时间?
- 1.5 硬件和软件在什么意义上是等效的?在什么意义上又是不等效的?试举例说明。
- 1.6 试以实例说明计算机系统结构、计算机组成与计算机实现之间的相互关系与相互影响。
- 1.7 什么是透明性概念?对计算机系统结构,下列哪些是透明的?哪些是不透明的?
存储器的模 m 交叉存取;浮点数据表示;I/O 系统是采用通道方式还是 I/O 处理机方式;数据总线宽度;阵列运算部件;通道是采用结合型的还是独立型的;PDP-11 系列中的单总线结构;访问方式保护;程序性中断;串行、重叠还是流水控制方式;堆栈指令;存储最小编址单位;Cache 存储器。
- 1.8 从机器(汇编)语言程序员看,以下哪些是透明的?
指令地址寄存器;指令缓冲器;时标发生器;条件码寄存器;乘法器;主存地址寄存器;磁盘外设;先行进位链;移位器;通用寄存器;中断字寄存器。
- 1.9 下列哪些对系统程序员是透明的?哪些对应用程序员是透明的?
系列机各档不同的数据通路宽度;虚拟存储器;Cache 存储器;程序状态字;“启动 I/O”指令;“执行”指令;指令缓冲寄存器。
- 1.10 实现软件移植的主要途径有哪些?它们存在什么问题?适用于什么场合?
- 1.11 想在系列中发展一种新型号机器,你认为下列哪些设想是可以考虑的,哪些则是行不通的?为什么?
 - (1) 新增加字符数据类型和若干条字符处理指令,以支持事务处理程序的编译。
 - (2) 为增强中断处理功能,将中断分析由原来的 4 级增加到 5 级,并重新调整中断响应的优先次序。
 - (3) 在 CPU 和主存之间增设 Cache 存储器,以克服因主存访问速率过低而造成的系统性能瓶颈。
 - (4) 为解决计算误差较大的问题,将机器中浮点数的下溢处理方法由原来的恒置“1”法,改为增设用只读存储器存放下溢处理结果的查表舍入法。
 - (5) 为增加寻址灵活性和减少平均指令字长,将原来全部采用等长操作码的指令改成有 3 类不同码长的扩展操作码;并将源操作数寻址方式由原来的操作码指明改成增加一个如 VAX-11 那样的寻址方式位字段来指明。
 - (6) 将 CPU 与主存之间的数据通路宽度由 16 位扩到 32 位,以加快主机内部信息的传送。

(7) 为了减少使用公用总线的冲突,将单总线改为双总线。

(8) 把原来的 0 号通用寄存器改作为专用的堆栈指示器。

- 1.12 如果某一计算任务用向量方式求解比用标量方式求解要快 20 倍,称可用向量方式求解部分所花费时间占总的时间的百分比为可向量化百分比。请画出加速比与可向量化比例两者关系的曲线。
- 1.13 在题 1.12 中,为达到加速比 2,可向量化的百分比应为多少?
- 1.14 在题 1.12 中,为获得采用向量方式最大加速比的半值(即 10)时,所需可向量化的百分比为多少?
- 1.15 在题 1.12 中,如果某程序可向量化部分为 70%,硬件设计组认为可以通过加大工程投资,使向量处理速度加倍来进一步增加性能;而编译程序编写组认为只需设法增加向量工作方式的百分比就可使性能得到相同的提高,问:此时需使可向量化成分再增加多少百分比就可实现?你认为上述硬、软件两种方法中,哪一种方法更好?
- 1.16 某计算机的高速小容量存储器能存储 2 000 条指令。假设其中 10%的指令承担了 90%的指令访问且对这 10%的指令的使用是均匀的(即其中每条指令的执行时间相同)。如果要执行的某程序共有 50 000 条指令且已知其中的 10%是频繁使用的,则当该计算机执行该程序时,在高速小容量存储器中能访问到的指令会占多少百分比?
- 1.17 假设高速缓存 Cache 工作速度为主存的 5 倍,且 Cache 被访问命中的概率为 90%,则采用 Cache 后,能使整个存储系统获得多高的加速比?
- 1.18 设计指令存储器有两种不同方案:一是采用价格较贵的高速存储器芯片,另一种是采用价格便宜的低速存储芯片。采用后一方案时,用同样的经费可使存储器总线带宽加倍,从而每隔 2 个时钟周期就可取出 2 条指令(每条指令为单字长 32 位);而采用前一方案时,每个时钟周期存储器总线仅取出 1 条单字长指令。由于访存空间局部性原理,当取出 2 个指令字时,通常这 2 个指令字都要使用,但仍有 25%的时钟周期中,取出的 2 个指令字中仅有 1 个指令字是有用的。试问采用这两种实现方案所构成的存储器带宽为多少?
- 1.19 用一台 40MHz 处理机执行标准测试程序,它含的混合指令数和相应所需的时钟周期数如下:

指令类型	指令数	时钟周期数
整数运算	45 000	1
数据传送	32 000	2
浮点	15 000	2
控制传送	8 000	2

求有效 CPI、MIPS 速率和程序的执行时间。

- 1.20 某工作站采用时钟频率为 15MHz、处理速率为 10MIPS 的处理机来执行一个已知混合程序。假定每次存储器存取为 1 周期延迟、试问:
- (1) 此计算机的有效 CPI 是多少?

(2) 假定将处理机的时钟提高到 30MHz, 但存储器子系统速率不变。这样, 每次存储器存取需要两个时钟周期。如果 30% 指令每条只需要一次存储存取, 而另外 5% 每条需要两次存储存取, 还假定已知混合程序的指令数不变, 并与原工作站兼容, 试求改进后的处理机性能。

- 1.21 假设在一台 40MHz 处理机上运行 200 000 条指令的目标代码, 程序主要由四种指令组成。根据程序跟踪实验结果, 已知指令混合比和每种指令所需的指令数如下:

指令类型	CPI	指令混合比
算术和逻辑	1	60%
高速缓存命中的加载/存储	2	18%
转移	4	12%
高速缓存缺失的存储器访问	8	10%

(1) 计算在单处理机上用上述跟踪数据运行程序的平均 CPI。

(2) 根据(1)所得 CPI, 计算相应的 MIPS 速率。

- 1.22 已知四个程序在三台计算机上的执行时间(s)如下:

程序	执行时间(s)		
	计算机 A	计算机 B	计算机 C
程序 1	1	10	20
程序 2	1 000	100	20
程序 3	500	1 000	50
程序 4	100	800	100

假设四个程序中每一个都有 100 000 000 条指令要执行, 计算这三台计算机中每台机器上每个程序的 MIPS 速率。根据这些速率值, 你能否得出有关三台计算机相对性能的明确结论? 你能否找到一种将它们统计排序的方法? 试说明理由。

- 1.23 在 SUN SPARC2 工作站上, 对 SPEC Benchmark 进行测试, 获得了如下所示的速率值, 求出其算术、几何及调和平均值(以 MFLOPS 表示)

程序名	速率(MFLOPS)
GCC	10.7
Espresso	8.9
Spice2g6	8.3
DODUC	5.0
NASA7	8.7
Li	9.0
Eqntott	9.7
Matrix300	11.1
FPPPP	7.8
TOMCATV	5.6

- 1.24 假定你是一个计算机设计者,对高级语言结构的使用研究表明,过程调用是最常用的操作之一。你已设想一个优化设计方案,它能减少过程调用和返回所需的取/存指令次数。为了进行验证,对未加优化和已优化的方案进行实验测试,假定所使用的是相同的优化编译器。实验测得的结果如下:
- (1) 未优化方案的时钟周期比优化的快 5%;
 - (2) 未优化方案中的取/存指令数占总指令数的 30%;
 - (3) 优化方案中的取/存指令数比未优化的少 1/3。对于其他指令,两种方案的动态执行数没有变化;
 - (4) 所有指令,包括取/存,均只需 1 个时钟周期。
- 要求你定量地判断,哪一种设计方案的计算机工作速度更快。
- 1.25 浮点运算的典型测试程序 Whetstone 含有 79 550 次浮点运算,其中加法 37 530 次,减法 3 520 次,乘法 22 900 次,除法 11 400 次以及将整数转换成浮点数 4 200 次。Whetstone 中还调用了下列各种特殊函数:反正切 640 次,正弦 640 次,余弦 1 920 次,开方、指数和对数各 930 次,Whetstone 的一次迭代所要完成的浮点操作总数,可以通过包含上述特殊函数调用所需执行的浮点操作来加以计算,总数为 195 578 次,其中:加法 82 014 次,减法 8 229 次,乘法 73 220 次,除法 21 399 次,将整数转换成浮点数 6 006 次,比较 4 710 次。
- 现若让 Whetstone 在 SUN3/75 工作站运行,使用优化的 F77 编译器。SUN3/75 由 Motorola MC68020 处理器构成,主频 16.6MHz,它有一个浮点协处理器(假定该协处理器不含有完成上述特殊函数的专用指令)。SUN 编译器根据编译器的标志,可使用协处理器来完成浮点运算,也可使用软件例行程序来执行浮点运算。用协处理器执行 1 次 Whetstone 迭代需时 10.8s,而用软件例程执行需时 13.6s。假定协处理器的 CPI 为 10,而使用软件例程时的 CPI 经测试为 6。
- 要求用 MIPS 值来表示这两种方法的执行速度。
- 1.26 计算题 1.25 中每一种方法执行的总的指令数为多少?
- 1.27 在题 1.25 中,平均而言,用软件例程完成一次浮点运算需要用多少条整数指令?
- 1.28 计算题 1.25 中,SUN3/75 的原始 MFLOPS 值和正则化后的 MFLOPS 值以及各种指令正则化后的具体值。

第二章 指令系统

指令系统是计算机系统中软件与硬件分界面的一个主要标志。无论多么复杂、功能多么强大的软件,凡是能够在机器上直接运行的目标程序都是由一条条机器指令组成的。在计算机系统的设计和使用过程中,硬件设计人员采用各种手段实现指令系统,而软件设计人员则使用这些指令系统编制各种各样的系统软件和应用软件,用这些软件来填补硬件的指令系统与人们习惯的使用方式之间的语义差距。因此,可以说,指令系统是软件设计人员与硬件设计人员之间的一个主要分界面,也是他们之间互相沟通的一座桥梁。在计算机系统的设计过程中,指令系统的设计是非常关键的,它必须由软件设计人员和硬件设计人员共同来完成。

计算机软件的发展非常迅速,特别是从第三代计算机之后。人们希望计算机能做更多的事,其功能更加强大,使用更加方便。然而,计算机的指令系统(也包括数据表示和寻址技术等)发展相当缓慢。几十年来,指令系统变化不大,或者说没有根本的改变。指令系统与软件之间的语义差距越来越大,因此,需要用软件来填补的东西也就越来越多,软件设计的任务变得越来越繁重,开发出来的软件越来越复杂。

这一章主要介绍指令系统及与指令系统直接相关的数据表示和寻址技术等方面的内容。由于数据表示、寻址技术和指令系统是系统结构的核心内容,在《计算机组成原理》、《计算机操作系统》、《汇编语言程序设计》等课程中都已涉及到这些内容。在本课程中,主要从计算机系统的分析和设计的角度介绍这部分内容,同时也介绍一些这方面的最新研究成果。

2.1 数据表示

本节主要介绍一些新的数据表示方法,如自定义数据表示等,也介绍一些数据表示方面的新的研究成果,如浮点数表示方面的研究成果等。一些常用的数据表示方法,如定点数、逻辑数、浮点数、字符、字符串、堆栈等,已经在其他课程中学过了。

2.1.1 数据表示与数据类型

在计算机系统中,数据的类型各种各样,有文件、图、表、树、阵列、队列、链表、栈、向量、串、实数、整数、布尔数、字符等。计算机体系结构首先要研究的一个内容就是:在所有这些数据类型中,哪些用硬件实现,哪些用软件实现,并研究它们的实现方法等。

数据表示研究的是计算机硬件能够直接识别、可以被指令系统直接调用的那些数据类型。数据表示是数据类型中最常用、也是相对比较简单、用硬件实现相对比较容易的几种,如定点数(整数)、逻辑数(布尔数)、浮点数(实数)、十进制数、字符、字符串、堆栈和向量等。

数据结构研究的是面向系统软件,面向应用领域所需要处理的各种数据类型,研究这些数据类型的逻辑结构和物理结构之间的关系,并给出相应的算法。除了数据表示之外的所有数据类型,一般来说都是数据结构要研究的内容。因此,数据表示和数据结构都是数据类型的子集。确定哪些数据类型用数据表示实现,哪些数据类型用数据结构实现,实质上是软、硬件的主要分界面之一,也是计算机系统设计中软件与硬件的取舍问题。

如何确定数据表示这个子集是计算机系统结构设计人员要解决的难题之一。从原理上讲,计算机系统只要有了最简单的数据表示,如定点数表示,就能用软件实现其他各种各样的数据类型,包括很复杂的数据类型。例如,能够用定点运算的指令编写的子程序来实现浮点运算,实现逻辑运算,实现十进制运算,实现字符运算,模拟堆栈运算等。当然,这种系统的性能可能很差。相反,如果把许多很复杂的数据类型都用数据表示来实现,系统的硬件成本就会很高。

确定哪些数据类型用数据表示来实现的原则主要有三个,一是缩短程序的运行时间,二是减少 CPU 与主存储器之间的通信量,三是这种数据表示的通用性和利用率。下面举两个例子来说明。

例 2.1 如果用定点数据表示实现浮点运算,处理机的运算速度要降低两个数量级。

如果用一台定点运算速度为每秒 1 千万次的计算机做科学计算,它的实际运算速度将低于每秒十万次。这是因为,当计算机系统中没有浮点数据表示时,通常要用子程序来实现浮点运算。用定点运算指令来实现 32 位的浮点运算时,平均要执行 100 条以上的指令。CPU 与主存储器之间的通信量也将增加 100 多倍。尽管增加浮点数据表示后硬件的复杂度要增加许多,但是,由于浮点数据表示的通用性好,利用率高,在以科学计算为主的计算机系统中,设置浮点数据表示是必不可少的。

例 2.2 实现 $A=A+B$, A 和 B 均为 200×200 的矩阵。

如果在没有向量数据表示的计算机系统上实现,一般需要 6 条指令,其中有 4 条指令要循环 4 万次。因此,CPU 与主存储器之间的通信量为:

取指令 $2+4 \times 40\,000$ 条,

读或写数据 $3 \times 40\,000$ 个,

共要访问主存储器 $7 \times 40\,000$ 次以上。

如果在有向量数据表示的计算机系统上实现,只需要一条指令。从而减少了 CPU 与主存储器之间的通信量:少取指令 $4 \times 40\,000$ 次,程序执行时间缩短了一半以上。

随着计算机系统的发展,数据表示也在不断地上移。例如,在目前的计算机系统中,字符串数据表示、向量数据表示、堆栈数据表示等已经普遍使用。有些很复杂的数据表示,如图、表等数据表示也开始在某些计算机系统中出现。

另外还应该指出,对于一些复杂的数据类型,如果用数据表示来实现,硬件的代价可能非常大,然而,如果用硬件给以适当的支持,或者说,用软件和硬件相结合的方法来实现,效果会很好。例如,用字节编址和字节运算指令来支持字符串数据表示。用变址寻址方式来支持向量数据表示等。

因此,在设计计算机系统时,对于数据类型,系统结构设计者首先要做的是:确定哪些数据类型全部用硬件实现,即数据表示;哪些数据类型用软件实现,即数据结构;哪些数据

类型可由硬件给予适当的支持,即由软件和硬件共同来实现,并确定软件与硬件的适当比例关系。

2.1.2 浮点数据表示

早期的计算机只有定点数据表示。这种计算机的优点是硬件结构比较简单,但有三个明显的缺点。

第一个缺点是编程困难,程序设计人员必须首先确定机器小数点的位置,并把所有参与运算的数据的小数点都对齐到这个位置上,然后机器才能正确进行运算。也就是说,编程人员首先要把参与运算的数据扩大或缩小某一个倍数后送入机器,等运算结果出来后再恢复到正确的数值。

第二个缺点是表示数的范围小,例如,一台 16 位字长的计算机所能表示的整数的范围只有 -32 768 到 32 767。从另一个角度看,为了能表示两个大小相差很大的数据,需要有很长的机器字长。例如:

太阳的重量大约是: 0.2×10^{34} 克,

一个电子的重量大约是: 0.9×10^{-27} 克,

两者相差在 10^{61} 以上。

若用定点数据表示: $2^x > 10^{61}$,

解得: $x > 203$ 位;

再加上精度的要求,例如,要求精度不低于 10 进制 7 位,

则有: $2^{-x} < 10^{-7}$,

解得: $x > 23$ 位;

总共需要: $203 + 23 = 230$ 位。

第三个缺点是数据存储单元的利用率往往很低。例如,为了把小数点的位置定在数据最高位前面,必须把所有参与运算的数据至少都除以这些数据中的最大数,只有这样才能把所有数据都化成纯小数,因而会造成很多数据有大量的前置零,从而浪费了许多数据存储单元。

为了解决上述三个问题,现代的大部分计算机都引入了浮点数据表示方式。在《汇编语言程序设计》课中,大家已经学习了浮点数的格式及其用法,在《计算机组成原理》课中,已经学习了浮点数的运算方法(加、减、乘、除等)及运算器的工作原理等,本书将重点介绍浮点数据的分析和设计方法。

计算机中的浮点数来源于数学中的实数,但两者又有许多本质的不同,如表 2.1 所示。

表 2.1 实数与浮点数的比较

	表示范围	表示精度	唯一性
实数	无限	连续	不冗余
浮点数	有限	不连续	冗余

浮点数表示方式要研究的核心内容是数据字长与这种数据表示方式的表数范围、表数精度和表数效率之间的关系。因为在计算机系统中数据是要用物理的存储部件存放的,因此,总是希望尽量减少数据的存储量,包括缩短数据的字长。而在通用计算机系统中,数据的字长一般定义为字节的整倍数,目前多为二进制的 32 位或 64 位。所以,实际上浮点数表示方式要研究的关键问题是:在数据字长已经确定的前提下,研究各种浮点数表示方式的表数范围、表数精度、表数效率及它们之间的关系等,并且企图寻找到一种具有最大表数范围、最高表数精度和最优表数效率的浮点数表示方式。

2.1.2.1 浮点数的表数范围

一个浮点数 N 可以用如下方式表示:

$$N = m \times r_e^e, \quad \text{其中: } e = r_e^e,$$

一种浮点数据表示方式需要有 6 个参数来定义。

两个数值:

m : 尾数的值,还包括尾数采用的码制(原码或补码)和数制(小数或整数);

e : 阶码的值,一般采用移码(又称偏码、增码、余码等)或补码,整数;

两个基:

r_m : 尾数的基,通常有二进制、四进制、八进制、十六进制和十进制等;

r_e : 阶码的基,在目前见到的所有浮点数据表示方式中, r_e 均为 2。

两个字长:

p : 尾数长度,要特别注意,这里的 p 不是指尾数的二进制位数,当 $r_m=16$ 时,每 4 个二进制位表示一位尾数;

q : 阶码长度,由于阶码的基通常为 2,因此,在一般情况下, q 就是阶码部分的二进制位数。

要注意:这里给出的 p 和 q 均不包括符号位。如没有特别说明,本书中见到的所有 p 和 q 也都不包括符号位。

一种浮点数表示方式如图 2.1 所示,这也是浮点数在数据存储单元中的存放方式。把尾数符号放在最高位的原因是为了判别正、负方便,也有一些浮点数表示方式把尾数符号与尾数放在一起的。



注: m_f 为尾数的符号位, e_f 为阶码的符号位, e 为阶码的值, m 为尾数的值。

图 2.1 浮点数在数据存储单元中的存放方式

研究浮点数表示方式的主要目的是用尽量短的字长(主要是阶码字长 q 与尾数字长 p 的和)实现尽可能大的表数范围和尽可能高的表数精度。根据这一目的,上述 6 个参数中只有 3 个参数是浮点数表示方式要研究的主要对象,它们是,尾数基值 r_m 、阶码字长 q 和尾数字长 p ,而另外 3 个参数与浮点数的表数范围和表数精度基本无关。阶码基值 r_e 通

常取为 2,在目前见到的计算机系统中已经成为定论,因为在以二进制为基本计算单位的计算机系统中阶码采用其他进位制没有任何好处。阶码的值 e 通常采用整数、移码表示,只有极少数机器中采用补码表示。尾数 m 在多数计算机中用纯小数表示,只有少数机器采用整数表示,而尾数的码制有原码和补码两种,然而,尾数 m 的表示方法与浮点数的表数范围和表数精度基本无关,它主要与 +、-、×、÷ 等运算方法和运算部件的设计等有关,这些在《计算机组成原理》课中已经学过。

下面根据浮点数与实数的三个主要区别来分析浮点数的主要性质,它们是:浮点数的表数范围、表数精度和表数效率。

在计算机中,数据总是要用存储部件来存放的,而计算机的存储部件的字长都是有限的,因此,任何一种浮点数表示方式所能表示的浮点数的个数和范围都是有限的。

在尾数采用原码、纯小数,阶码采用移码、整数的浮点数表示方式中,规格化浮点数 N 的表数范围是:

$$r_m^{-1} \cdot r_e^{-r_g} \leq |N| \leq (1 - r_m^{-p}) \cdot r_e^{q-1} \quad (2.1)$$

例如, $p=23, q=7, r_m=r_e=2$, 尾数用原码、纯小数表示,阶码用移码、整数表示,规格化浮点数 N 的表数范围是:

$$\frac{1}{2} \cdot 2^{-27} \leq |N| \leq (1 - 2^{-23}) \cdot 2^{27-1}$$

即 $2^{-129} \leq |N| \leq (1 - 2^{-23}) \cdot 2^{127} \quad (2.2)$

从表 2.2 中可以看出,规格化浮点数的最大正数值是由尾数的最大正数值与阶码的最大正数值组合而成的,而最小正数值是由尾数的最小正数值与阶码的最小负数值组合而成的。在负数区间,规格化浮点数的最大负数值是由尾数的最大负数值与阶码的最小负数值组合而成的,最小负数值是由尾数的最小负数值与阶码的最大正数值组合而成的,这一点也可以从图 2.2 所示的数轴上看到。

表 2.2 尾数用原码、纯小数时规格化浮点数的表数范围

表数范围	规格化尾数	阶码	规格化浮点数
最大正数 (N_{\max})	$1 - r_m^{-p}$	$r_e^q - 1$	$(1 - r_m^{-p}) \cdot r_e^{q-1}$
最小正数 (N_{\min})	r_m^{-1}	---	$r_m^{-1} \cdot r_e^{-r_g}$
最大负数 ($-N_{\max}$)	$-r_m^{-1}$	---	$-r_m^{-1} \cdot r_e^{-r_g}$
最小负数 ($-N_{\min}$)	$-(1 - r_m^{-p})$	$-r_e^q$	$-(1 - r_m^{-p}) \cdot r_e^{q-1}$

注: r_m 为尾数的基, r_e 为阶码的基, p 为尾数长度, g 为阶码长度。

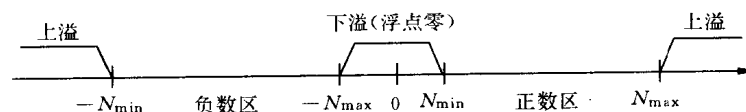


图 2.2 浮点数在数轴上的分布情况

浮点数表示法中,阶码通常要用移码表示的主要原因是:从上面看到的,规格化浮点数的最大负数($-N_{\max}$)和取小正数(N_{\min})都与通过阶码的最小负数值有关,而在除移码之外的其它所有码制中,最小负数都不是全0(包括符号位)。如果浮点零与机器零不一致,对软件设计和硬件设计都会造成许多麻烦。

当尾数用补码表示时,正数区间的表数范围与尾数采用原码时完全相同,而负数区间的表数范围为:

$$-r_m^{r_e-1} \leq N \leq -(r_m^{-1} + r_m^{-p}) \cdot r_m^{-r_e} \quad (2.3)$$

例如,尾数用补码、纯小数表示,阶码用移码、整数表示, $p=6, q=6, r_m=16, r_e=2$,规格化浮点数 N 在正数区间的表数范围是:

$$\frac{1}{16} 16^{-2^6} \leq N \leq (1 - 16^{-6}) \cdot 16^{2^6-1}$$

$$\text{即} \quad 16^{-65} \leq N \leq (1 - 16^{-6}) \cdot 16^{63} \quad (2.4)$$

在负数区间的表数范围是:

$$-16^{63} \leq N \leq -\left(\frac{1}{16} + 16^{-6}\right) \cdot 16^{-64}$$

表 2.3 尾数采用补码、纯小数时规格化浮点数的表数范围

表数范围	规格化尾数	阶码	规格化浮点数
最大正数	$1 - r_m^{-p}$	$r_e^q - 1$	$(1 - r_m^{-p}) \cdot r_m^{r_e^q-1}$
最小正数	r_m^{-1}	—	$r_m^{-1} \cdot r_m^{-r_e^q}$
最大负数	$-(r_m^{-1} + r_m^{-p})$	—	$-(r_m^{-1} + r_m^{-p}) \cdot r_m^{-r_e^q}$
最小负数	-1	$-r_e^q$	$-r_m^{r_e^q-1}$

注: r_m 为尾数的基, r_e 为阶码的基, p 为尾数长度, q 为阶码长度。

从上面的分析中可以看到,在尾数采用纯小数表示的情况下,由于尾数的绝对值只能在 0.5 到 1 之间变化,因此,规格化浮点数的表数范围主要与阶码的长度 q 和尾数的基值 r_m 有关。这时,能表示的绝对值最大的浮点数可近似为:

$$|N_{\max}| = r_m^{r_e^q} \quad (2.5)$$

假设有两种浮点数表方式示 $F1$ 和 $F2$:

$F1$: 尾数的基值为 2, 阶码的长度为 $q1$;

$F2$: 尾数的基值为 r_m , 阶码的长度为 $q2$, 并设 $k = \lceil \log_2 r_m \rceil$;

两种浮点数表示方式表数范围的比值为:

$$T = (r_m, q) = \frac{N_{\max 2}}{N_{\max 1}} = \frac{r_m^{2^{q2}}}{2^{2^{q1}}} = \frac{2^{k \cdot 2^{q2}}}{2^{2^{q1}}} = 2^{k \cdot 2^{q2} - 2^{q1}} \quad (2.6)$$

从(2.6)式中得到: 浮点数阶码的字长 q 每增加一个二进制位, 所能表示的阶码最大

值就增加一倍。当尾数的基值从 2 增加到 r_m 时,所能表示的阶码最大值就增加 $\lceil \log_2 r_m \rceil$ 倍,而浮点数的表数范围根据其阶码的增加按 2 的指数增加。

从上面的两个例子中可以看到,(2.2)式所表示的浮点数的二进制总字长为:

$$L1 = q1 + p1 + 1 + 1 = 7 + 23 + 1 + 1 = 32$$

不等式(2.5)所表示的浮点数的二进制字长为:

$$L2 = q2 + k p2 + 1 + 1 = 6 + 24 + 1 + 1 = 32$$

两种浮点数表示方式的总的二进制字长是相同的,都是 32 个二进制位,但它们的表数范围相差很大。因为 $q1=7, r_{m1}=2, q2=6, r_{m2}=16, k=\log_2 16=4$,所以第二种浮点数表示方式的表数范围与第一种浮点数表示方式的比值为:

$$T = 2^{k \cdot 2^{q2} - 2^{q1}} = 2^{4 \cdot 2^6 - 2^7} = 2^{128}$$

2.1.2.2 浮点数的表数精度

表数精度也称为表数误差,浮点数存在表数精度的根本原因是由于浮点数的不连续性造成的,因为任何一种浮点数表示方式的字长总是有限的。如果字长为 32 位,则这种浮点数表示方式所能表示的浮点数的个数最多是 2^{32} 个(大约 43 亿个),而数学中的实数是连续的,它有无穷多个,因此,一种浮点数表示方式能表示的浮点数的个数只是实数中很少的一部分,即它的一个子集,我们称之为这种浮点数表示方式的浮点数集。

浮点数集 F 的表数误差可以这样定义,令 N 是浮点数集 F 内的任一给定实数,而 M 是 F 中最接近 N ,且被用来代替 N 的浮点数,则绝对表数误差(absolute representation error)为:

$$\delta = |M - N|$$

相对表数误差(relative representation error)为:

$$\delta = \left| \frac{M - N}{N} \right|$$

由于在同一种浮点数表示方式中,规格化浮点数的尾数有效位长度是确定的,因此,规格化浮点数的相对表数误差是确定的,又由于规格化浮点数在数轴上的分布是不均匀的,因此它的绝对表数误差是不确定的。这一点与定点数表示方式正好相反,定点数表示方式的绝对误差是确定的,而其相对误差是不确定的。以下主要介绍相对表数误差,如没有特别说明,本书中所讨论的表数误差都是指相对表数误差。

浮点数表数误差产生的直接原因有两个,一个是两个浮点数 a 和 b 都在某种浮点数表示方式的浮点集内,而 a 与 b 的运算(通常指 $+$ 、 $-$ 、 \times 、 \div 等)结果却可能不在这个浮点集(不是溢出)内,因此必须用这个浮点集内与它最接近的某个浮点数来表示,从而也就造成了表数误差。另一个是在将数据从十进制转化为二进制、四进制、八进制或十六进制时可能产生误差。下面我们分别举例说明。

如果一种浮点数表示方式的 $q=1, p=2, r_m=2, r_e=2$,尾数用原码、纯小数表示,阶码用移码、整数表示。这种浮点数表示方式所能表示的浮点数具体可以参见图 2.3 和表 2.4。

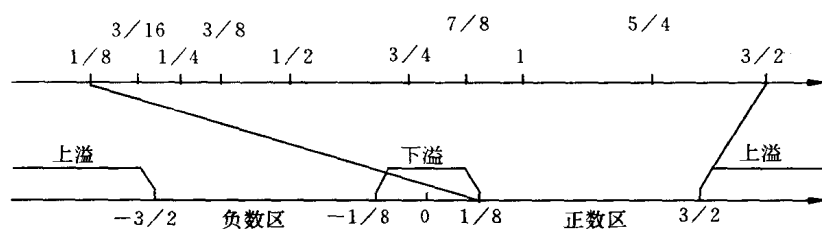


图 2.3 一种浮点数在数轴上的分布

表 2.4 一种浮点数表示方式所能表示的全部浮点数

阶码		阶码 ($q=1, r_m=2$)			
		1 (1.1)	0 (1.0)	-1 (0.1)	-2 (0.0)
尾数	0.75 (0.11)	3/2	3/4	3/8	3/16
	0.5 (0.10)	1	1/2	1/4	1/8
	-0.5 (1.10)	-1	-1/2	-1/4	-1/8
	-0.75 (1.11)	-3/2	-3/4	-3/8	-3/16

注：尾数用原码、纯小数表示，阶码用移码、整数表示。括号中的数字是二进制，其余为十进制。

若有两个浮点数： $a_1=1/2, b_1=3/4$ 都在所定义的浮点集内，而它们相加的结果： $a_1+b_1=5/4$ ，则不在这个浮点数集内，为此必须用该浮点数集内的最靠近 $5/4$ 的浮点数 1 或 $3/2$ 来代替，从而产生绝对表数误差：

$$\delta = |5/4 - 3/2| = 1/4 \quad \text{或} \quad \delta = |5/4 - 1| = 1/4,$$

相对表数误差为：

$$\delta = \left| \frac{5/4 - 3/2}{5/4} \right| = 1/5 \quad \text{或} \quad \delta = \left| \frac{5/4 - 1}{5/4} \right| = 1/5$$

若： $a_2=3/8, b_2=1/2$ ，两数也都在所定义的浮点集内，而它们相加的结果： $a_2+b_2=7/8$ 则不在这个浮点数集内，为此必须用该浮点数集内的最靠近 $7/8$ 的浮点数 $3/4$ 或 1 来代替，从而产生表数误差。

$$\delta = \left| \frac{7/8 - 3/4}{7/8} \right| = 1/7$$

产生浮点数表数误差的另一个原因是：原始数据从外部进入计算机的过程中，通常要将数据从十进制转化为二进制、四进制、八进制或十六进制等。因为物理数据存储单元的长度总是有限的，从而可能产生表数误差。例如，把十进制数 0.1 输入二进制的计算机中，出现循环小数，即：

$$\begin{aligned} 0.1_{(10)} &= 0.000110011001100\cdots_{(2)} \\ &= 0.0121212\cdots_{(4)} \\ &= 0.06146314\cdots_{(8)} \\ &= 0.1999\cdots_{(16)} \end{aligned}$$

用有限长度的尾数是不能精确表示这个数的，于是就产生了一个如何处理不能表示

的尾数低位部分的问题,通常要进行舍入处理。舍入的方法有很多种,如十进制中的“4舍5入”就是其中的一种,各种不同舍入方法产生的表数误差也不同,比较详细的讨论见(2.1.2.6)节。这里要特别注意的是,为了在有限的尾数字长内获得尽量高的表数精度,舍入必须在规格化之后进行。规格化时要以 $\lceil \log_2 r_m \rceil$ 为单位进行移位,例如,十进制数0.1进行规格化后变成:

$$\begin{aligned} 0.1_{(10)} &= 0.110011001100\cdots_{(2)} \times 2^{-3} \\ &= 0.121212\cdots_{(4)} \times 4^{-1} \\ &= 0.6146314\cdots_{(8)} \times 8^{-1} \\ &= 0.1999\cdots_{(16)} \times 16^0 \end{aligned}$$

实际上,数字在机器内部都是用二进制表示的。如果尾数用二进制12位,阶码用二进制3位,尾数用原码、小数表示,阶码用移码、整数表示,并且另外各有一个符号位,表示结果如表2.5。

表 2.5 十进制数 0.1 的多种浮点数表示方式

尾数的基值	阶符	阶 码	尾符	尾 数
$r_m=2$	0	101	0	1100110011001100
$r_m=4$	0	111	0	0110011001100110
$r_m=8$	0	111	0	1100110011001100
$r_m=16$	1	000	0	0001100110011001

表 2.5 中的尾数都是在各自的尾数基值 r_m 下的规格化表示,从表中可以看出,对于不同的尾数基值 r_m ,舍入时要处理的尾数低位部分也是不一样的,舍入后的最终结果也不会相同。在一般情况下,随着尾数基值 r_m 的增大,规格化后的二进制尾数中的前置“0”的个数也就越多。只有在尾数基值 $r_m=2$ 时,才能保证在规格化后的尾数中不出现前置“0”,从而获得最高的表数精度。

从上面的分析中我们已经知道,规格化浮点数的表数精度主要与尾数基值 r_m 和尾数长度 p 有关。在一般情况下,认为规格化尾数最后1位的精确度是一半,这样,规格化浮点数的表数精度可以表示如下:

$$\delta(r_m, p) = \frac{1}{2} r_m^{-(p-1)} \quad (2.7)$$

当 $r_m=2$ 时,有:

$$\delta(2) = \frac{1}{2} \cdot 2^{-(p-1)} = 2^{-p} \quad (2.8)$$

当 $r_m>2$ 时,为了便于与二进制进行比较,我们用 p_2 表示二进制的尾数字长,于是有如下关系:

$$P = \lceil \frac{p_2}{\log_2 r_m} \rceil, \quad r_m = 2^{\log_2 r_m}$$

代入(2.7)式,有:

$$\delta(r_m) = 2^{\lceil \log_2 r_m \rceil - 1} \cdot 2^{-p_2} \quad (2.9)$$

把(2.9)与(2.8)进行比较,由于(2.9)中的 p_2 就是(2.8)中的 p ,因此,我们可以得出:

结论:当浮点数的尾数长度(指二进制位数)相同时,尾数基值 r_m 取 2 具有最高的表数精度。当尾数基值 $r_m > 2$ 时,浮点数的表数精度与 $r_m = 2$ 相比将损失 $2^{\lceil \log_2 r_m \rceil - 1}$ 倍,即相当于尾数减少了 $(\lceil \log_2 r_m \rceil - 1)$ 个二进制位。

实际上,得出上述结论是很显然的,因为,当尾数基值 $r_m = 2$ 时,规格化浮点数的尾数肯定没有前置 0,全部尾数都是有效位,而当尾数基值 $r_m > 2$ 时,规格化浮点数的尾数最多可能有 $(\lceil \log_2 r_m \rceil - 1)$ 个前置 0,所以,当尾数基值 $r_m > 2$ 时,浮点数的表数精度与 $r_m = 2$ 相比将损失 $2^{\lceil \log_2 r_m \rceil - 1}$ 倍。例如,当尾数基值 $r_m = 16$ 时,规格化浮点数的尾数可能有 3 个前置 0,因此其表数精度与 $r_m = 2$ 时相比将损失 2^3 倍,即相当于尾数减少了 3 个二进制位。

2.1.2.3 浮点数的表数效率

通常把尾数最高位为非零的浮点数称为规格化浮点数,只有一个特例,就是机器零,机器零的尾数和阶码都是 0,但它也是一个规格化浮点数。

由于在同一种浮点数表示方式中,规格化浮点数具有最长的尾数有效位数,即能够保证尾数的绝对值为最大,当尾数用小数表示时,其绝对值在 $\frac{1}{r_m}$ 到 1 之间,所以,规格化浮点数的表数精度是最高的。

在采用浮点数表示方式的计算机中,一般规定存放在存储部件中的浮点数、进入运算部件的浮点数、从运算部件输出的浮点数都必须是规格化浮点数,只有在运算过程中才可能出现非规格化的浮点数。

在浮点数运算过程中,当出现非规格化浮点数时,必须通过左规格化或右规格化操作来把它变换成规格化浮点数。如果尾数采用小数表示,当出现尾数绝对值小于 $\frac{1}{r_m}$ (尾数的最高位为 0) 时,要把尾数左移,每次左移 $\lceil \log_2 r_m \rceil$ 个二进制位,同时把阶码减 1,直到尾数的绝对值大于 $\frac{1}{r_m}$ 时为止,当出现尾数绝对值大于 1 时,要把尾数右移,每次左移 $\lceil \log_2 r_m \rceil$ 个二进制位,同时把阶码加 1,直到尾数的绝对值小于 1 时为止。

由于浮点数表示方式中有可能出现非规格化浮点数,因此,浮点数制是一种冗余数制 (redundant number system),这一点,与数学中的实数是不同的。

为了提高数据的信息利用率,总是希望非规格化浮点数的个数尽可能少,为此,必须研究浮点数表示方式的表数效率问题。

浮点数表示方式的表数效率定义为:

$$\eta = \frac{\text{可表示的规格化浮点数的个数}}{\text{全部浮点数个数}} = \frac{2 \cdot (r_m - 1) \cdot r_m^{p-1} \cdot 2 \cdot r_e^q + 1}{2 \cdot r_m^p \cdot 2 \cdot r_e^q}$$

分子中,第一个 2 表示尾数的正、负数各半, $(r_m - 1)$ 表示尾数最高位的编码种数,这里去掉了一个 0, r_m^{p-1} 表示尾数除最高位之外其他位的编码种数,第二个 2 表示由于阶码的符号位使编码种数增加一倍, r_e^q 表示阶码的所有编码种数,分子中的最后一个 1 表示机器零,因为机器零也属于规格化浮点数;分母中,第一个 2 表示尾数的正、负数各半, r_m^p 表示尾数的全部编码种数,第二个 2 表示由于阶码的符号位使编码种数增加一倍, r_e^q 表示阶码的所有编码种数。

上式经过化简,并忽略掉机器零,有:

$$\eta(r_m) = \frac{r_m - 1}{r_m} \quad (2.10)$$

从(2.10)中看出,浮点数的表数效率主要与尾数的基值有关。

当尾数基值为 2 时,浮点数的表数效率为:

$$\eta(2) = \frac{2 - 1}{2} = 50\%$$

因此,尾数基值取 2 时,浮点数的信息利用率很低。这时,只有尾数最高一个二进制位为 1 的浮点数是规格化浮点数,尾数最高一个二进制位为 0 的浮点数是非规格化浮点数。

当尾数基值 $r_m > 2$ 时,浮点数的表数效率与尾数基值 $r_m = 2$ 时相比提高的倍数可计算如下:

$$T = \frac{\eta(r_m)}{\eta(2)} = 2 - \frac{1}{2^{\lceil \log_2 r_m \rceil - 1}},$$

变化范围在 1 与 2 之间,并且随着 r_m 的增大,这个比值就越来越接近 2。

例如,当尾数基值 $r_m = 16$ 时,浮点数的表数效率为:

$$\eta(16) = \frac{16 - 1}{16} = 94\%$$

这是因为当尾数基值 $r_m = 16$ 时,只要尾数的最高 4 个二进制位不全为 0,所表示的浮点数都是规格化浮点数,只有当最高 4 个二进制位全为 0 时,所表示的浮点数才是非规格化浮点数。

尾数基值 $r_m = 16$ 与 $r_m = 2$ 相比,浮点数的表数效率提高了:

$$T = \frac{\eta(16)}{\eta(2)} = 1.875 \text{ 倍。}$$

2.1.2.4 浮点数尾数基值的选择

从上一节的分析中看到,浮点数的表数范围、表数精度和表数效率三个主要性质都与尾数的基值 r_m 有关。因此,本节专门介绍在设计一种浮点数表示方式时,尾数基值 r_m 应当如何选择。

实际上,在表示浮点数的 6 个参数中,只有尾数基值 r_m 、尾数长度 p 和阶码长度 q 这三个参数与浮点数的表数范围、表数精度和表数效率三个主要性质有关。另外,在通用计

算机中,浮点数的总字长通常是确定的,目前多为 32 或 64 个二进制位,而在总字长中,除了尾数符号和阶码符号各占去一个二进制位之外,其余就是尾数长度 p 与阶码长度 q 之和。所以,设计一种浮点数表示方式的关键是要解决两个问题,一是在浮点数的总字长(主要是尾数长度 p 与阶码长度 q 之和)给定的情况下,如何选择尾数基值 r_m ,使浮点数的表数范围最大、表数精度和表数效率最高;二是在尾数基值 r_m 确定之后,如何根据表数范围和表数精度的要求具有确定尾数长度 p 与阶码长度 q 。本节主要介绍前一个问题,下一节介绍后一个问题。

从(2.5)中看到,浮点数的表数范围主要与阶码长度 q 和尾数基值 r_m 有关,从(2.7)中看到,表数精度主要与尾数长度 p 和尾数基值 r_m 有关。我们假设有两种浮点数表示方式 $F1$ 和 $F2$,它们二进制字长相同,尾数都用原码或补码、小数表示,阶码都用移码、整数表示,阶码的基均为 2,而尾数的基不同。

浮点数表示方式 $F1$: 尾数基值 $r_{m1}=2$,尾数长度 $p1$,阶码长度 $q1$,二进制字长: $L1=p1+q1+2$ 。

浮点数表示方式 $F2$: 尾数基值 $r_{m2}=2^k$ (k 为正整数),尾数长度 $p2$,阶码长度 $q2$,二进制字长: $L2=k \cdot p2+q2+2$ 。

由 $F1$ 与 $F2$ 的二进制字长相同,即 $L1=L2$,得:

$$p1+q1=k \cdot p2+q2 \quad (2.11)$$

下面,首先介绍浮点数字长和表数范围一定时,尾数基值 r_m 与表数精度的关系。

由上一节的(2.5)知道: $F1$ 的表数范围是:

$$|N_{1\max}| = 2^{2^{q1}},$$

同样, $F2$ 的表数范围是:

$$|N_{2\max}| = (2^k)^{2^{q2}}, \text{即: } |N_{2\max}| = 2^{k \cdot 2^{q2}}$$

因为 $F1$ 与 $F2$ 的表数范围相同,所以 $F1$ 与 $F2$ 的阶码应当相等:

$$2^{q1} = k \cdot 2^{q2}$$

两边分别取以 2 为底的对数,得到:

$$q1 = q2 + \log_2 k \quad (2.12)$$

把(2.12)式代入(2.11)式,得到:

$$p1 = k \cdot p2 - \log_2 k \quad (2.13)$$

由上一节的(2.7)知道: $F1$ 的表数精度是:

$$\delta_1 = \frac{1}{2} \cdot 2^{1-p1} \quad (2.14)$$

把(2.13)代入(2.14)得到:

$$\delta_1 = \frac{1}{2} \cdot 2^{1-k \cdot p2 - \log_2 k}$$

$F1$ 的表数精度是:

$$\delta_2 = \frac{1}{2} \cdot 2^{k \cdot (1-p2)}$$

为了考察 $F1$ 与 $F2$ 的表数精度之间的关系,取它们的比值:

$$T = \frac{\delta_2}{\delta_1} = 2^{k-\log k-1} \quad (2.15)$$

从(2.15)式中可以看出,只有 $k=1$ (表示尾数基值 $r_m=2$) 或 $k=2$ (表示尾数基值 $r_m=4$) 时,比值 $T=1$; 当 k 取其它任何值时,比值 T 都大于 1 ($T>1$ 表示 $F2$ 的精度低于 $F1$)。

因此,可以得出结论:在浮点数的字长和表数范围一定时,尾数基值 r_m 取 2 或 4 具有最高的表数精度。

在得出这一结论之后,我们再从另一个角度来分析,在浮点数的字长和表数精度一定时,尾数基值 r_m 与表数范围之间的关系。

由 $F1$ 与 $F2$ 的表数精度相同得到:

$$\frac{1}{2} \cdot 2^{1-p_1} = \frac{1}{2} \cdot (2^k)^{1-p_2}$$

即: $p_1 = k p_2 - k + 1 \quad (2.16)$

把(2.16)代入(2.11)得到:

$$q_1 = q_2 + k - 1 \quad (2.17)$$

再根据(2.5)分别计算 $F1$ 和 $F2$ 的表数范围,并把(2.17)式代入:

$F1$ 的表数范围:

$$2^{2^{q_1}} = 2^{2^{q_2+k-1}} = 2^{2^{q_2}} \cdot 2^{k-1}$$

$F2$ 的表数范围:

$$(2^k)^{2^{q_2}} = 2^{2^{q_2} \cdot k}$$

先假设 $F2$ 的表数范围大于 $F1$ 的表数范围,则应该有 $F2$ 阶码的最大值要大于 $F2$ 阶码的最大值:

$$2^{q_2} \cdot k > 2^{q_2} \cdot 2^{k-1}$$

即: $k > 2^{k-1}$

这个不等式在正整数定义域内没有解,也就是说:不存在比 $F1$ 的表数范围更大的浮点数表示方式,只有当 $k=1$ (表示尾数基值 $r_m=2$) 或 $k=2$ (表示尾数基值 $r_m=4$) 时, $F2$ 阶码的最大值等于 $F2$ 阶码的最大值。

因此,可以得出另一个结论:在浮点数的字长和表数精度一定时,尾数基值 r_m 取 2 或 4 具有最大的表数范围。

综合上面两个结论,可以得出一个新的重要推论:在浮点数的字长确定之后,尾数基值 r_m 取 2 或 4 具有最大的表数范围和最高的表数精度。

这一结论在浮点数表示方式的研究中具有非常重要的作用,下面,举一例子来说明这一结论。

IBM 370 系列计算机的短浮点数表示方式如图 2.4(a)所示,尾数基值 $r_m=16$,尾数字长为 16 进制 6 位,即二进制 24 位,阶码基值 $r_m=2$,阶码字长 6 位,尾数用原码、小数表示,阶码用移码、整数表示。

这种浮点数表示方式的表数精度为:

$$\cdot 48 \cdot$$

$$\delta = \frac{1}{2} \cdot 16^{-(6-1)} = 2^{-21}$$

表数范围是：

$$|N_{\max}| = 16^{2^6} = 2^{256}$$

若设计另外一种浮点数表示方式，取尾数基值 $r_m=2$ ，并要求与尾数基值 $r_m=16$ 时有同样的表数精度，则有：

$$\frac{1}{2} \cdot 2^{-(p-1)} = 2^{-21}$$

解得 $p=21$ ，因为总字长仍为 32 位，所以阶码字长 $q=32-21-2=9$ 位，这种浮点数表示方式如图 2.4(b) 所示。它的表数范围是：

$$|N_{\max}| = 2^{2^9} = 2^{512}$$

从这个例子看出，在浮点数的总字长和表数精度要求一定的情况下，尾数基值 $r_m=2$ 比尾数基值 $r_m=16$ 有更大的表数范围。这个结论也可以这样来解释：取尾数基值 $r_m=16$ 时的表数精度与尾数基值 $r_m=2$ 时的表数精度相比损失了 3 个二进制位，而为了使尾数基值 $r_m=2$ 时的表数范围能够与尾数基值 $r_m=16$ 时相同，阶码的字长只要在原来 6 位的基础上再增加 2 位。

$$16^{2^q} = 2^{q+2}$$

因此，为了达到与 IBM 370 系列机同样的表数范围和表数精度，浮点数的总字长只要 31 个二进制位即可。

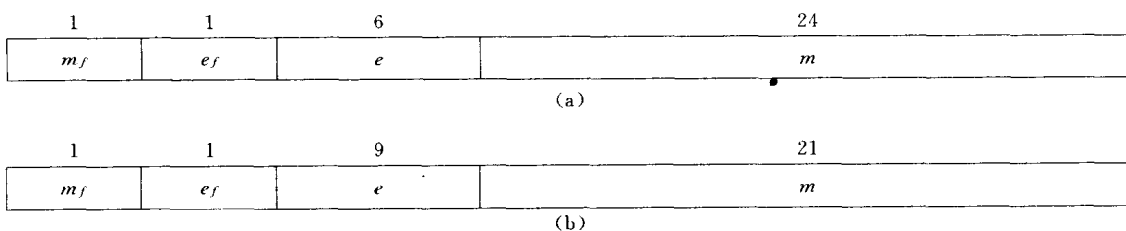


图 2.4 两种浮点数表示方式

发生上述情况的根本原因是：当浮点数的尾数基值从 $r_m=2$ 增大时，表数精度是按尾数字长的指数关系变低，而表数范围却按阶码字长的指数的指数关系变大，因此，为了保持同样的表数精度和表数范围，损失的尾数字长大于节省的阶码字长，从而造成浮点数的总字长增加。

上面的结论告诉我们，从浮点数的表数范围和表数精度看，尾数基值 r_m 取 2 或 4 最好，但从 (2.10) 式中得到，当尾数基值 $r_m=2$ 时，表数效率最低，只有 50%。为了在尾数基值 r_m 取 2 时提高浮点数的表数效率，目前在许多机器中采用了隐藏位表数方法。具体做法是：因为在尾数基值 $r_m=2$ 时，规格化浮点数尾数的最高位一定是 1（如果尾数用补码表示，规格化浮点数尾数最高位一定与尾数符号位相反），所以，浮点数在存储和传送的过

程中,尾数最高位可以不表示出来,只在计算时恢复这一个隐藏位,或采用某种方法对运算结果进行修正。

在采用隐藏位表数方法之后,当取尾数基值 $r_m=2$ 时,浮点数的表数效率能够达到最大值 100%。如果取尾数基值 $r_m=4$,就不能采用隐藏位表数方法,这时的浮点数的表数效率为 75%。

综上所述,浮点数的尾数基值 r_m 取 2,并采用隐藏位表数方法是最佳的浮点数表示方式。这种浮点数表示方式能做到表数范围最大、表数误差最小、表数效率最高。

目前,IBM 公司的 IBM 360 系列机、IBM 370 系列机、IBM 4300 系列机等,浮点数的尾数基值 $r_m=16$ 。Burroughs 公司的 B6700、B7700 等大型计算机,浮点数的尾数基值 $r_m=8$ 。DEC 公司的 VAX-11 和 Alpha 等计算机,CDC 公司的 CDC6600、CYBER70 等大型计算机及应用最广泛的 Intel 公司的 x86 系列机等,浮点数均采用尾数基值 $r_m=2$ 。浮点数的尾数基值不取 2 的原因可能是设计人员在早期对浮点数的本质研究不深,后来虽然弄明白了尾数基值取 2 最好,但是为了系列计算机具有兼容性,也只能如此了。

2.1.2.5 浮点数格式的设计

定义一种浮点数表示方式需要确定 6 个参数,这 6 个参数的确定原则如下:

1. 尾数 m 的数制和码制。主要从运算简单、表数直观等方面来考虑。目前多数机器用小数、原码表示,只有早期的一些没有硬件乘除法指令的机器采用补码表示。尾数采用原码制表示,虽然加减法比采用补码表示复杂,但乘除法要比采用补码表示简单得多,而且,采用原码表示非常直观。

另外,尾数采用小数表示能简化运算,特别是乘除法运算。

2. 阶码 e 的数制和码制,目前一般机器都采用整数、移码(或称增码、偏码、余码等)表示。阶码采用移码表示的主要原因是:使浮点零与机器零一致。如果浮点零与机器零不同,对机器硬件和软件的设计都会产生许多麻烦。例如,要判断运算结果是否为零必须作特殊的处理。然而阶码采用移码表示后,在做浮点乘除法运算时,由于阶码要进行加减运算,移码的加减法运算要比补码复杂。

由于阶码主要是用来扩大浮点数的表数范围的,因此,它必须用整数表示。

3. 尾数的基值 r_m ,在上一节中已经得出结论:在浮点数的总字长一定的情况下,尾数的基值 r_m 选择 2,浮点数具有最大的表数范围和最高的表数精度。如果再采用隐藏位表示方法,则这种浮点数表示方法同时又具有最高的表数效率。

4. 阶码的基值 r_e ,在一般通用计算机中,浮点数阶码的基值都取 2,这是因为,在目前的一般计算机中,基本存储单元都是二态的。另外,浮点数中的阶码起指数的作用,主要用来扩大表数范围,阶码采用其它进位制与采用二进制相比并不能扩大浮点数的表数范围,因此,浮点数中的阶码采用二进制是必然的。

另外两个参数是尾数字长 p 和阶码字长 q 。如何确定尾数字长 p 和阶码字长 q ,这是本节要解决的主要问题。

目前,多数计算机的短浮点数是 32 位(二进制),长浮点数是 64 位,尽管在早期计算机中,浮点数的字长有 16 位、24 位、36 位、48 位等多种,但这些计算机目前已经很少使

用。在浮点数总字长中,除了尾数符号和阶码符号各占一个二进制位之外,其余都由尾数字长 p 和阶码字长 q 占据。

在前一节的分析中已经介绍过,在浮点数表示方式中,尾数字长 p 主要影响表数精度,阶码字长 q 主要影响表数范围,而浮点数的表数范围和表数精度是浮点数的两个最主要的特性。

浮点数的表数范围和表数精度是从计算机的实际应用领域中提出来的,不同的应用领域对浮点数的表数范围和表数精度可能有不同的要求。以下,主要介绍在浮点数的表数范围和表数精度给定的情况下,如何具体确定浮点数的尾数字长 p 和阶码字长 q ,即设计浮点数的格式。

根据大多数计算机的实际情况,我们假定:浮点数的尾数用原码、小数表示,阶码用移码、整数表示,尾数基值 $r_m = 2$,阶码基值 $r_e = 2$,并根据实际应用要求给出表数范围不小于 N (N 为能表示的最大的正数),表数精度不低于 δ ,并且要求尾数和阶码都正、负对称。

由表数范围的要求,得到:

$$2^{2^q-1} > N$$

解这个不等式:

$$2^q > \frac{\log N}{\log 2} + 1$$

得到阶码字长:

$$q > \frac{\log\left(\frac{\log N}{\log 2} + 1\right)}{\log 2} \quad (2.18)$$

根据表数精度的要求,得到:

$$\frac{1}{2} \cdot 2^{-(p-1)} < \delta$$

解这个不等式:

$$2^{-p} < \delta$$

得到围数字长:

$$p > \frac{-\log \delta}{\log 2} \quad (2.19)$$

由(2.18)和(2.19)两个不等式计算出浮点数的尾数字长 p 和阶码字长 q ,再加上一个尾数符号位和一个阶码符号位就构成了一种浮点数格式。应当注意的是在满足这两个不等式的前提下,通常要适当调整尾数字长 p 和阶码字长 q 的取值,使浮点数的总字长达到一个合理的数值。

下面我们举一个具体的例子。

要求设计一种浮点数格式,其表数范围不小于 10^{-37} 至 10^{37} ,正、负数对称,表数精度不低于 10^{-16} 。

根据表数范围的要求,用(2.18)不等式计算:

$$q > \frac{\log\left(\frac{\log N}{\log 2} + 1\right)}{\log 2} = \frac{\log\left(\frac{\log 10^{37}}{\log 2} + 1\right)}{\log 2} = 6.95$$

取阶码字长 $q=7$ 。

根据表数精度的要求,用(2.19)不等式计算:

$$p > \frac{-\log \delta}{\log 2} = \frac{-\log 10^{-16}}{\log 2} = 53.2$$

如果取尾数字长 $p=54$, 则浮点数的总字长为: $p+q+2=54+7+2=63$, 离 8 的整数倍 64 还差一位, 这一位可以加到尾数字长 p 中用于提高浮点数的表数精度, 也可以加到阶码字长 q 中用于提高浮点数的表数范围。在这里, 暂且把它加到尾数字长 p 中以提高浮点数的表数精度。这样设计出来的浮点数的格式如图 2.5 所示, 尾数 55 位, 阶码 7 位, 阶码符号和尾数符号各一位。

1	1	7	55
m_f	e_f	e	m

注: m_f 为尾数的符号位, e_f 为阶码的符号位, e 为阶码的值, m 为尾数的值。

图 2.5 一种浮点数的表示方式

在浮点数的尾数用原码、小数表示, 阶码用移码、整数表示, 尾数基值 $r_m=2$, 阶码基值 $r_e=2$, 阶码字长 $q=7$, 尾数字长 $p=55$, 尾数符号和阶码符号各一位, 总字长为 64 位。图 2.5 给出的这种浮点数表示方式的各项主要性能如下:

能表示的最大尾数值: $(1-r_m^{-p})=(1-2^{-55})$, 即尾数数值部分的所有 55 个二进制位全部都为 1;

绝对值最小的尾数值: $\frac{1}{r_m}=\frac{1}{2}$, 尾数数值部分除最高一个二进制位为 1 之外, 其余 54 个二进制位全部为 0;

能表示的最大阶码: $r_e^q-1=2^7-1=127$, 包括阶码符号位在内的所有 8 个二进制位全部为 1;

能表示的最小阶码: $-r_e^q=-2^7=-128$, 包括阶码符号位在内的所有 8 个二进制位全部为 0;

最大正数: $(1-r_m^{-p}) \cdot r_m^{q-1}=(1-2^{-55}) \cdot 2^{2^7-1}=(1-2^{-55}) \cdot 2^{127}=1.70 \times 10^{38}$;

最小正数: $\frac{1}{r_m} \cdot r_m^{-r_e^q}=\frac{1}{2} \cdot 2^{-2^7}=2^{-129}=1.47 \times 10^{-39}$;

最大负数: $-\frac{1}{r_m} \cdot r_m^{-r_e^q}=-\frac{1}{2} \cdot 2^{-2^7}=-2^{-129}=-1.47 \times 10^{-39}$;

最小负数: $-(1-r_m^{-p}) \cdot r_m^{r_e^q-1}=(1-2^{-55}) \cdot 2^{-(2^7-1)}=-(1-2^{-55}) \cdot 2^{127}=-1.70 \times 10^{38}$;

表数精度: $\delta=\frac{1}{r_m} \cdot r_m^{-(p-1)}=\frac{1}{2} \cdot 2^{-(55-1)}=2^{-55}=2.78 \times 10^{-17}$;

浮点零: 由于阶码采用移码表示, 且正、负数对称, 因此, 浮点零与机器零相同, 即 64 个二进制位全为 0;

表数效率: $\eta=\frac{r_m-1}{r_m}=\frac{2-1}{2}=50\%$, 如果采用隐藏位表数方法, 则表数效率能达到最高 $\eta=100\%$;

能表示的浮点数个数: $2(r_m-1)r_m^{p-1}r_e^{q+1}+1=2 \cdot (2-1) \cdot 2^{55-1} \cdot 2^{7+1}+1=9.22 \times 10^{18}$ 。

2.1.2.6 浮点数的舍入处理

浮点数要进行舍入处理的原因是:由于任何一种浮点数表示方式的尾数字长总是有限的,因此,可能发生如下两个问题:一是在把日常使用的十进制实数转化为计算机内部的浮点数时,其有效位长度可能超过给定的尾数字长,这时必须舍去多余的部分。二是两个规格化浮点数的加、减、乘、除运算结果,其尾数长度可能超过给定的尾数字长。例如,两个浮点数相乘时,其乘积的尾数字长是给定浮点数字长的两倍;两个浮点数相加时,首先要进行对阶,阶码小的那个浮点数的尾数要右移,移出去的部分就超过了给定的尾数字长;同样,两个浮点数相除时,可能由于除不尽而要多上商几位,从而商的尾数长度可能超过给定的尾数字长。

舍入处理要解决的问题是:假设某种浮点数表示方式给定的尾数字长为 p 位,而两个尾数长度为 p 位的规格化浮点数的加、减、乘、除运算结果,其尾数的规格化长度为 $p+g$ 位;同样,一个日常使用的十进制实数在转化为计算机内部的浮点数时,其尾数的规格化长度也可能是 $p+g$ 位;浮点数舍入要解决的主要问题就是把这规格化尾数的 $p+g$ 位处理成只有 p 位。

舍入方法的主要性能标准是:本身的误差小,积累误差小,容易实现。由于舍入方法有很多种,它们各有自己的优点和缺点,其出发点及应用场合也有所不同,因此要根据实际情况进行比较和选择。

在进行舍入处理时要注意的问题是:必须先规格化,然后再舍入,否则舍入是没有意义的;在计算积累误差时,要同时考虑到正数区和负数区的情况;在对负数作舍入处理时,要注意不同的码制。

以下,主要针对原码制介绍 5 种常用的舍入方法,由于原码的正数区和负数区是对称的,因此其舍入规则比较简单,最后再讨论补码制和反码制的舍入方法。另外,在计算误差和积累误差时,假设浮点数尾数用小数表示。

1. 恒舍法

恒舍法又称截断法、必舍法等,这是一种最容易实现的舍入方法。

假定舍入前规格化尾数的长度为 $p+g$ 位, p 是尾数有效字长, g 是有效字长 p 位之外的代码长度。恒舍法的舍入规则是:无论 g 位长度的代码是什么,一律把它舍去,只保留有效字长 p 位代码作为尾数,而且不作任何修改。

设舍入前规格化尾数在正数区的值 m_x 为:

$$m_x = \pm 0. \text{xxx} \cdots \text{x} 000 \cdots 0 \sim \pm 0. \text{xxx} \cdots \text{x} 111 \cdots 1$$

其中, $\text{xxx} \cdots \text{xx}$ 表示尾数在 p 位有效字长内的数值, $000 \cdots 0 \sim 111 \cdots 1$ 表示超过有效字长的代码,长度是 g 位。舍入后规格化尾数在正数区的值 m 为:

$$m = \pm 0. \text{xxx} \cdots \text{x}$$

由于超过尾数有效字长 p 位之外的 g 位代码全部被舍掉,在正数区,尾数的实际数值变小,因此是负误差,在负数区,尾数的实际数值变大,因此是正误差。

在正数区的误差范围为:

$$\delta = m - m_x = -2^{-p}(1 - 2^{-g}) \sim 0$$

在负数区的误差范围为：

$$\delta = m - m_x = 0 \sim 2^{-p}(1 - 2^{-g})$$

绝对值最小的误差是 0, 这时被舍掉的 g 位代码为全 0; 绝对值最大的误差是 $2^{-p}(1 - 2^{-g})$, 即被舍掉的 g 位代码为全 1。

由于在正数区和在负数区的误差符号相反, 因此有利于减小积累误差。如果尾数在正数区和在负数区各种代码出现的概率完全相同, 则总的积累误差为 0, 即积累误差达到完全平衡。但是, 当正数区和负数区分别考虑时, 恒舍法的积累误差很大, 这是恒舍法的最大缺点。

浮点数的积累误差通常定义为规格化尾数所有取值的误差之和。

恒舍法在正数区的积累误差计算如下：

$$\sigma = (-0.0\cdots000|0\cdots000) + \cdots + (-0.0\cdots000|1\cdots111)$$

$$\sigma = -2^{-p-1}(2^g - 1)$$

符号“|”左边是尾数有效字长之内的数值, 共有 p 位, “|”右边是尾数有效字长之外的数值, 共有 g 位。同样, 在负数区的积累误差为 $\sigma = +2^{-p-1}(2^g - 1)$, 与正数区相比, 绝对值相等, 符号相反, 这有利于积累误差的平衡。

从上式中可以看出, 恒舍法的积累误差与尾数的有效字长 p 和有效字长之外的长度 g 有关。当 $p=g$ 时, 积累误差 σ 可近似为 2^{-1} , 即 50% 的误差, 这时的浮点数已经没有精度可言了。

恒舍法的舍入规则和误差情况见表 2.6。

表 2.6 恒舍法的舍入规则和误差情况

		尾数有效字长 p 位	有效字长之外 g 位	误差情况
正 数 区	舍入前	0. xxx...xx	00...000	$\delta=0$
		0. xxx...xx	00...001	$\delta=-2^{-p-g}$
		0. xxx...xx	00...010	$\delta=-2^{-p-g+1}$
	
		0. xxx...xx	11...111	$\delta=-2^{-p}(1-2^{-g})$
	舍入后	0. xxx...xx		$\sigma=-2^{-p-1}(2^g-1)$
负 数 区	舍入前	-0. xxx...xx	00...000	$\delta=0$
		-0. xxx...xx	00...001	$\delta=+2^{-p-g}$
		-0. xxx...xx	00...010	$\delta=+2^{-p-g+1}$
		-0. xxx...xx
		-0. xxx...xx	11...111	$\delta=+2^{-p}(1-2^{-g})$
	舍入后	-0. xxx...xx		$\sigma=+2^{-p-1}(2^g-1)$

恒舍法的主要优点是实现起来非常容易。但是, 由于通用计算机在 1 秒钟内要进行几千万次以上的运算, 积累误差是必须要考虑的, 因此, 在运算精度要求比较高的应用场合, 不宜采用恒舍法。

2. 恒置法

恒置法又称恒置 $\frac{r}{2}$ 法(r 是尾数的基值),或恒置1法(当尾数基值取2时),或冯·诺依曼法(Von Neumann rounding)。

恒置法的实现难度仅次于恒舍法。

恒置法的舍入规则是:把规格化尾数有效字长 p 位的最低一位置成 $\frac{r}{2}$,而不管超过有效字长之外的 g 位代码是什么。当尾数基值取2时,把尾数有效位的最低一位置成1,当尾数基值取16时,把尾数有效位的最低一位置成8。

若舍入前规格化尾数 m_x 共有 $p+g$ 位:

$$m_x = \pm 0. \text{xxx} \cdots \text{x0} | 000 \cdots 00 \sim \pm 0. \text{xxx} \cdots \text{x0} | 111 \cdots 11,$$

符号“|”左边是尾数有效字长之内的数值,共有 p 位,“|”右边是尾数有效字长之外的数值,共有 g 位。则舍入后规格化尾数 m 有 p 位,其值为:

$$m = \pm (0. \text{xxx} \cdots \text{x0} + 2^{-p}) = \pm 0. \text{xxx} \cdots \text{x1},$$

把有效字长之外的 g 位全部舍掉,并在有效字长之内的数值上加了 2^{-p} 。

若舍入前规格化尾数 m_x 为:

$$m_x = \pm 0. \text{xxx} \cdots \text{x1} | 000 \cdots 0 \sim \pm 0. \text{xxx} \cdots \text{x1} | 111 \cdots 1,$$

则舍入后规格化尾数 m 为:

$$m = \pm 0. \text{xxx} \cdots \text{x1},$$

把有效字长之外的 g 位全部舍掉,与恒舍法相同。

恒置法在正数区的误差范围为:

$$\delta = m - m_x = -2^{-p}(1 - 2^{-g}) \sim +2^{-p},$$

在负数区的误差范围为:

$$\delta = m - m_x = -2^{-p} \sim +2^{-p}(1 - 2^{-g}),$$

恒置法在正数区的舍入规则及误差情况见表2.7。

表 2.7 恒置法在正数区的舍入规则及误差情况

正数区	尾数有效字长 p 位	有效字长之外 g 位	误差情况
舍入前	0. xxx...xx0	00...000	$+2^{-p}$ ——误差积累
	0. xxx...xx0	00...001	$+2^{-p}(1 - 2^{-g})$
	0. xxx...xx0	00...010	$+2^{-p}(1 - 2^{-g+1})$

	0. xxx...xx0	11...111	$+2^{-p+g}$
	0. xxx...xx1	00...000	0
	0. xxx...xx1	00...001	-2^{-p+g}

	0. xxx...xx1	11...110	$-2^{-p}(1 - 2^{-g+1})$
	0. xxx...xx1	11...111	$-2^{-p-1} - 2^{-p-2}$
舍入后	0. xxx...xx1		积累误差 $\sigma = 2^{-p}$

从表 2.7 中看出,恒置法在正数区的绝大部分积累误差被平衡掉了,只有 $\sigma = +2^{-p}$ 是真正的积累误差。同样,在负数区的积累误差是 $\sigma = -2^{-p}$ 。与恒舍法相比,恒置法的积累误差要小得多,而且仅与尾数字长 p 有关,与被舍去的长度 g 无关。另外,正数区与负数区的积累误差正好绝对值相等,符号相反,因此,正负区的共同积累误差是平衡的。

恒置法的主要缺点是表数精度比较低,这是由于尾数的最低位被恒置成了 $\frac{r}{2}$,因此损失了一位精度。其主要优点是实现起来比较容易,在正数区和负数区的积累误差都比较小,而且能达到平衡。目前,恒置法被广泛地应用在各种计算机系统中。

3. 下舍上入法

在日常使用的十进制中称为 4 舍 5 入法,在二进制中称为 0 舍 1 入法,在 16 进制中称为 7 舍 8 入法。

下舍上入法的舍入规则是:以规格化尾数有效字长 p 位之外的 g 位代码的中间值为界,小于这个中间值的则舍,大于或等于这个中间值的则入。

若舍入前规格化尾数 m_r 共有 $p+g$ 位,

$$m_r = \pm 0. \text{xxx} \cdots \text{x} | 000 \cdots 0 \sim \pm 0. \text{xxx} \cdots \text{x} | 011 \cdots 1,$$

符号“|”左边是尾数有效字长之内的数值,共 p 位,右边是尾数有效字长之外的数值,共 g 位。则舍入后规格化尾数 m 有 p 位,其值为:

$$m = \pm 0. \text{xxx} \cdots \text{x},$$

这时的做法与恒舍法相同,即把尾数有效字长之外的 g 位全部舍掉,仅保留有效字长之内的 p 位数值。

若舍入前规格化尾数 m_r 为:

$$m_r = \pm 0. \text{xxx} \cdots \text{x} | 100 \cdots 0 \sim \pm 0. \text{xxx} \cdots \text{x} | 111 \cdots 1,$$

则舍入后规格化尾数 m 为:

$$m = \pm (0. \text{xxx} \cdots \text{x} + 2^{-p}),$$

先把尾数有效字长之外的 g 位全部舍掉,再在尾数上加 2^{-p} (在负数区是减 2^{-p})。这里要特别注意的是:由于做了一次加法运算,尾数有可能要溢出,为此要再次进行右规格化,当采用硬件实现时,可能要增加一个节拍,从而加长了整个浮点运算的时间,这是下舍上入法不能被普遍采用的主要原因。

实际上,下舍上入法在判别是舍还是入时,只要看规格化尾数有效字长之外的 g 位代码的最高一个二进制位即可,可以不管 g 位代码的其它位,也不用管尾数基值是什么,若 g 位代码的最高一个二进制位为 0 则舍,为 1 则入,这一点在用硬件实现下舍上入法时是很有用的。

参照表 2.8,下舍上入法在正数区的误差为:

$$\delta = m - m_r = -2^{-p-1}(1 - 2^{-g+1}) \sim +2^{-p-1},$$

在负数区的误差为:

$$\delta = m - m_r = -2^{-p-1} \sim +2^{-p-1}(1 - 2^{-g+1}),$$

与恒置法相比,尾数精度提高了一位。

下舍上入法的积累误差在正数区是 $\sigma = +2^{-p-1}$,在负数区是 $\sigma = -2^{-p-1}$ 。与恒舍法相

比, 积累误差小了一位, 这是增加了实现的难度换来的。同样, 积累误差仅与尾数的有效字长 p 有关, 与被舍去的长度 g 无关, 而且正数区与负数区的积累误差正好完全平衡。

下舍上入法的舍入规则及误差情况见表 2.8。

表 2.8 上舍下入法的舍入规则及误差情况

正数区	尾数有效字长 p 位	有效字长之外 g 位	误差 σ
舍入前	0. xxx...xx	00...000	0
	0. xxx...xx	00...001	-2^{-p-k}
	0. xxx...xx	00...010	-2^{-p-k+1}

	0. xxx...xx	01...111	$-2^{-p-1}(1-2^{-k})$
	0. xxx...xx	10...000	$+2^{-p-1}$ —积累误差
	0. xxx...xx	10...001	$+2^{-p-1}(1-2^{-k})$

	0. xxx...xx	11...110	$+2^{-p-k+1}$
	0. xxx...xx	11...111	$+2^{-p-k}$
舍入后	0. xxx...xx	最高位为 0	积累误差 $\sigma = +2^{-p-1}$
	0. xxx...xx + 2^{-p}	最高位为 1	

下舍上入法的主要优点是精度高, 积累误差小, 两者都比恒置法提高了一位, 相当于增加了一位字长, 而且在正数区和负数区的积累误差能达到完全平衡。主要缺点是实现起来比较困难, 因为在舍入之后可能要再次进行右规格化。目前, 下舍上入法已很少实际使用, 主要用于用软件实现的浮点运算中。

4. R * 舍入法

在大型、巨型计算机中, 或在一些很大的科学计算问题中, 需要一种积累误差能够完全平衡, 精度又很高的舍入方法。

在上述三种舍入方法中, 只有下舍上入法具有精度高, 而且在正数区和负数区分别考虑时, 积累误差比较小的优点。因此, 只要对下舍上入法稍加修改, 使它的积累误差达到完全平衡, 这将是一种理想的舍入方法。

从表 2.9 中可以看出, 把 0. xxx...xx | 00...00 (符号“|”左边是尾数有效字长之内的代码, 共 p 位, 右边是尾数有效字长之外的代码, 共 g 位, 下同) 舍为 0. xxx...xx 是没有误差的, 把 0. xxx...xx | 00...01 舍为 0. xxx...xx 与 0. xxx...xx | 11...11 入为 0. xxx...xx + 2^{-p} 两种情况产生的误差正好绝对值大小相等而符号相反, 因此积累误差是平衡的, 同样, 把 0. xxx...xx | 00...010 舍为 0. xxx...xx 与 0. xxx...xx | 11...10 入为 0. xxx...xx + 2^{-p} 两种情况产生的积累误差也是平衡的, ……直到把 0. xxx...xx | 01...11 舍为 0. xxx...xx 与 0. xxx...xx | 10...01 入为 0. xxx...xx + 2^{-p} 两种情况产生的积累误差都是平衡的, 只有 0. xxx...xx | 10...00 这一种情况产生的误差得不到平衡, 这是下舍上入法存在积累误差的根本原因。为了使这种积累误差能达到完全平衡, R * 舍入法对这种特殊情况进行单独处理, 改用恒置法, 对二进制来说是恒置 1 法, 即把 0. xxx...x0 | 10...00 入为 0. xxx...x1, 产生的误差为 $+2^{-p-1}$, 而把 0. xxx...x1 | 10...00 舍为 0. xxx...x1, 产生的误差为 -2^{-p-1} , 这

两种情况所产生的积累误差达到完全平衡。

R * 舍入法的舍入规则及误差情况见表 2.9 所示。

表 2.9 R * 舍入法的舍入规则及误差情况

舍入方法	舍入前($p+g$)位	舍入后(p 位)	误差情况
下舍上入法	0. xxx...xx 00...00	0. xxx...xx	0
下舍上入法	0. xxx...xx 00...01	0. xxx...xx	-2^{-p-k}
下舍上入法	0. xxx...xx 00...010	0. xxx...xx	-2^{-p-k+1}
....
下舍上入法	0. xxx...xx 01...11	0. xxx...xx	$-2^{-p-1}(1-2^{-k+1})$
恒置 1 法	0. xxx...x0 10...00	0. xxx...x1	$+2^{-p-1}$
恒置 1 法	0. xxx...x1 10...00	0. xxx...x1	-2^{-p-1}
下舍上入法	0. xxx...xx 10...01	0. xxx...xx + 2^{-p}	$+2^{-p-1}(1-2^{-k+1})$
....
下舍上入法	0. xxx...xx 11...10	0. xxx...xx + 2^{-p}	$+2^{-p-k+1}$
下舍上入法	0. xxx...xx 11...11	0. xxx...xx + 2^{-p}	$+2^{-p-k}$

负数区的舍入规则及误差分析方法与正数区完全相同,只要把原来是负误差的地方改为正误差,而原来是正误差的地方改为负误差,总的积累误差也是完全平衡的。

R * 舍入法在正数区和负数区的误差范围是相同的,都是:

$$\delta = m - m_x = -2^{-p-1} \sim +2^{-p-1}$$

综上所述,R * 舍入法是一种完全没有积累误差的舍入方法,而且精度也很高,但是,实现起来非常复杂,它的复杂主要表现在:首先要判断尾数有效字长之外的所有 g 位代码是否为 10...00,然后根据判断结果决定是采用下舍上入法,还是采用恒置法。如果采用了下舍上入法,还要判断舍入后的尾数是否溢出,若溢出,要再次进行右规格化。目前只有少数以向量计算为主的巨型计算机(如美国 Burroughs 公司和依里诺大学计算机科学系合作设计的 BSP 科学处理机)采用 R * 舍入法。

5. 查表法

又称 ROM 舍入法,PLA 舍入法等。

查表法继承了下舍上入法精度高、积累误差小的优点,同时又克服了它实现起来比较困难的缺点,是一种比较理想的舍入方法。

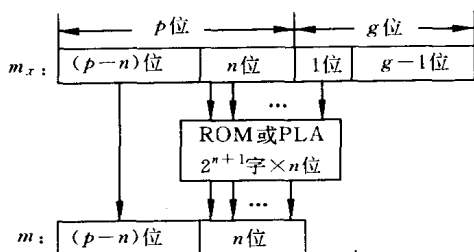


图 2.6 查表法的原理框图

查表法的原理框图如图 2.6 所示, m_x 为舍入前的规格化尾数,共 $p+g$ 位, p 是有效字长, g 是有效字长之外的代码长度。查表法需要一个只读存储器 ROM 或可编程逻辑阵列 PLA。用 m_x 有效字长的最低 n 位和有效字长之外的最高 1 位作为 ROM 或 PLA 的输入地址,读出 n 位代码作为舍入结果 m 的最低 n 位,而 m 的高 $p-n$ 位直接来源于 m_x 的最高 $p-n$ 位。在

实际使用中,ROM 或 PLA 的存储容量和输出位的位数可以有不同的选择,所存储的内容也可以根据不同的需要恰当安排,从而获得满意的舍入特性,使舍入误差和积累误差最小。

查表法的舍入规则及误差情况可参见表 2.10,在表中取 $n=2$,这是为了分析方便,当 n 取较大值时结果是一样的。ROM 或 PLA 有 3 位输入地址,来源于舍入前规格化尾数有效字长(p 位)的最低两位和被舍掉部分(g 位)的最高 1 位,存储器 8 个单元中存放的内容就是舍入后规格化尾数最低两位代码。

表 2.10 $n=2$ 时查表法的舍入规则及误差情况分析

ROM 地址	舍入前(p 位+ g 位)	舍入后(p 位)	误差情况
000	xx...x00 0xx...x	xx...x00	$-2^{-p-1}(1-2^{-g+1}) \sim 0$
001	xx...x00 1xx...x	xx...x01	$+2^{-p-g} \sim 2^{-p-1}$
010	xx...x01 0xx...x	xx...x01	$-2^{-p-1}(1-2^{-g+1}) \sim 0$
011	xx...x01 1xx...x	xx...x10	$+2^{-p-g} \sim 2^{-p-1}$
100	xx...x10 0xx...x	xx...x10	$-2^{-p-1}(1-2^{-g+1}) \sim 0$
101	xx...x10 1xx...x	xx...x11	$+2^{-p-g} \sim 2^{-p-1}$
110	xx...x11 0xx...x	xx...x11	$-2^{-p-1}(1-2^{-g+1}) \sim 0$
111	xx...x11 1xx...x	xx...x11	$+2^{-p}(1-2^{-g}) \sim -2^{-p-1}$

从表中可以看出,除最后一行(也可以说是最后两行)之外都采用下舍上入法,只有最后一行采用了恒舍法,这样可以避免再次规格化,因为在最后一行,尾数所有有效位上的代码可能为全 1,这时再加 2^{-p} 会造成尾数溢出。

查表法在正数区的误差范围是:

$$\delta = -2^{-p}(1-2^{-g}) \sim +2^{-p-1},$$

其绝对值最大的负误差来源于表 2.10 中的最后一行,由于表中其它行采用的是下舍上入法,因此,最大正误差与下舍上入法相同。同样,在负数区的误差范围为:

$$\delta = -2^{-p-1} \sim +2^{-p}(1-2^{-g}),$$

查表法在正数区的积累误差可以这样计算:

前 $2^{n+1}-2$ 行的积累误差是: $(2^n-1)2^{-p-1}$,

最后两行的积累误差是: $-2^{-p-1}(2^g-1)$,

总的积累误差是: $2^{-p-1}(2^n-2^g)$,

当 $n=g$ 时,积累误差完全平衡,这是因为最后两行的负积累误差与前面其它行的所有正积累误差完全抵消了。当 $n>g$ 时,积累误差为正,当 $n<g$ 时,积累误差为负,其主要原因是:查表法最后两行采用的是恒舍法,它的负积累误差与被舍掉的 g 位代码的长度有关,而前面的其它所有行采用的是下舍上入法,它的正积累误差与被舍掉的 g 位代码的长度无关。

查表法在负数区的积累误差与在正数区的分析方法相同,只要把+、-、>、<等符号反过来即可。

另外,由于查表法的积累误差与被舍掉的 g 位代码的长度有关,这对积累误差的平

衡是不利的,但是,我们可以通过精心设计 ROM 中所存储的内容,针对各种不同的应用场合使积累误差尽可能小。

查表法的主要优点是可以很方便地通过修改 ROM 或 PLA 中的内容来使积累误差达到完全平衡。由于 VLSI 技术的飞速发展,增加一个存储容量不大的 ROM 或 PLA 已经不成为问题,因此查表法是一种极有前途的舍入方法。

以上我们介绍了五种舍入方法,表 2.11 列出了这五种舍入方法的主要性能,包括在正数区的误差范围、积累误差和实现的难易程度等。

表 2.11 五种舍入方法的主要性能比较

舍入方法	在正数区的误差范围	正数区的积累误差	实现的难易程度
恒舍法	$-2^{-p}(1-2^{-g}) \sim 0$	$+2^{-p-1}(2^g-1)$	最简单
恒置法	$-2^{-p}(1-2^{-g}) \sim +2^{-p}$	$+2^{-p}$	很简单
下舍上入法	$-2^{-p-1}(1-2^{-g+1}) \sim +2^{-p-1}$	$+2^{-p-1}$	很复杂
R* 舍入法	$-2^{-p-1} \sim +2^{-p-1}$	0	最复杂
查表法	$-2^{-p}(1-2^{-g}) \sim +2^{-p-1}$	$+2^{-p-1}(2^n-2^g)$	一般

p 是规格化尾数有效位的字长, g 是尾数有效字长之外的长度, n 是查表法中 ROM 或 PLA 的字长

从表 2.11 中,我们可以得出如下结论:

(1) 恒置法虽有少量的积累误差,且损失一位精度,但由于实现起来很容易,已普遍在小型及微型机中使用。

(2) R* 舍入法是唯一一种积累误差能达到完全平衡的舍入方法,但由于实现起来非常复杂,仅在极少数对误差要求非常高的巨型计算机中才采用。

(3) 下舍上入法只有少量的积累误差,且精度也比较高,但实现起来很复杂,在用软件实现的算法中经常采用。

(4) 查表法实现比较容易,积累误差很小,而且可以通过改变 ROM 或 PLA 中的内容来修正积累误差,因此是一种很有前途的舍入方法。

下面我们简单介绍一下补码制和反码制中的舍入方法。

补码制中,恒舍法在负数区的积累误差与正数区相同,都是负误差,因此积累误差非常大,如果要像在原码制中那样,使负数区为正误差,则要把负数补码改用“恒入法”,实现起来非常困难,因此,在补码制中不宜采用恒舍法,其余四种舍入方法均可采用,而且舍入规则保持不变。

反码制中,只有恒舍法可以保持与原码制中相同的舍入方法不变;恒置法在负数区要改为恒置反码法,如恒置 1 法在负数区要改为恒置 0 法;下舍上入法在负数区要改为上舍下入法,如二进制中的 0 舍 1 入法在负数区要改为 1 舍 0 入法,且“入”的意思是做减法;查表法和 R* 舍入法与此类似,即负数区的舍入规则由正数区的规则经 0、1 交换和加、减交换后得到。经过这样修改后的舍入方法,对反码制具有与原码制相同的舍入特性,但是其实现的复杂性大大增加了,这也是反码制很少被采用的原因之一。

2.1.2.7 警戒位的设置方法

为了保证浮点数在运算和转换过程中的精度,在规定的尾数字长之外,运算器中的累加器需要另外增加的长度称为警戒位(guard digit or guard bit)。

例如,两个浮点数相加时,首先要进行对阶,阶码小的那个浮点数的尾数要右移,因此,有一部分代码要从规定字长的最低位移出去,如果把这部分代码保存下来,在对运算结果进行左规格化时,这部分保存下来的代码就可以左移进入尾数规定的字长之内,这样可以提高浮点数的运算精度。在浮点数的乘法、除法等运算过程中及日常用的十进制实数向浮点数转换的过程中也可能出现这样的问题。

举一个具体例子:

例 2.3 $0.10\ 000\ 000 \times 2^0 - 0.11\ 111\ 101 \times 2^{-1}$

在尾数数值部分取 8 位字长,没有警戒位时:

$$\begin{array}{rcl}
 & 0.10\ 000\ 000 \times 2^0 & \\
 \text{对阶:} & +1.10\ 000\ 001 \times 2^0 & \\
 \text{求和:} & 0.00\ 000\ 001 \times 2^0 & \\
 \text{左规:} & 0.10\ 000\ 000 \times 2^{-7} &
 \end{array}$$

如果尾数数值部分同样取 8 位字长,并设置一个警戒位时:

$$\begin{array}{rcl}
 & 0.10\ 000\ 000 \times 2^0 & \\
 \text{对阶:} & +1.10\ 000\ 001\ 1 \times 2^0 & \\
 \text{求和:} & 0.00\ 000\ 001\ 1 \times 2^0 & \\
 \text{左规:} & 0.11\ 000\ 000 \times 2^{-7} &
 \end{array}$$

没有警戒位时,运算结果为: $0.10\ 000\ 000 \times 2^{-7}$,有一个警戒位时,运算结果为: $0.11\ 000\ 000 \times 2^{-7}$,显然,后者的精度要比前者高得多。

我们再举一个例子,如果不设置警戒位,可能造成完全错误的运算结果。

例 2.4 $0.1\ 000 \times 2^{125} - 0.1\ 111 \times 2^{124}$

尾数数值部分取 4 位字长,并设置一个警戒位时:

$$\begin{array}{rcl}
 & 0.1\ 000 \times 2^{125} & \\
 \text{对阶:} & +1.1\ 000\ 1 \times 2^{125} & \\
 \text{求和:} & 0.0\ 000\ 1 \times 2^{125} & \\
 \text{左规:} & 0.1\ 000 \times 2^{121} &
 \end{array}$$

运算结果为: $0.1\ 000 \times 2^{121}$,是一个很大的数。但是,如果尾数数值部分同样取 4 位字长,不设置警戒位时,运算结果是 0,这显然是错误的。

$$\begin{array}{rcl}
 & 0.1\ 000 \times 2^{125} & \\
 \text{对阶:} & +1.1\ 000 \times 2^{125} & \\
 \text{求和:} & 0.0\ 000 \times 2^{125} &
 \end{array}$$

那么,警戒位应当如何设置?警戒位的长度应该设多少位?是否愈长愈好?

在浮点数中设置警戒位,不需要增加浮点数的字长。从上面的例子中已经看到,实际上只需要增加一个累加器(存放阶码小的那个浮点数的累加器)的长度,而其它数据寄存器和存储器的长度、运算器的长度等都不必增加。

下面,我们来分析警戒位的长度设置多少位最好。

这个问题必须从两个方面来分析,一个是警戒位的来源,另一个是它的用处。如果在一次运算过程中,警戒位只有来源而没有用处,那么即使设置了很长的警戒位也是没有用的,相反,如果警戒位只有用处而没有来源,设置警戒位也同样是没有什么用的,因此,只有有来源又有用处时才需要设置警戒位。

警戒位的用处只有两个:

1. 用于左规格化时移入尾数有效字长内。
2. 用于舍入。

警戒位的来源有以下几个方面:

1. 做加、减法时,因对阶从有效字长内移出去的部分。
2. 做乘法时,双倍字长乘积的低字长部分。
3. 做除法时,因没有除尽而多上商的几位。
4. 右规格化时移出有效字长的那部分。
5. 从十进制实数转换成二进制浮点数时,尾数超出有效字长的那部分。

警戒位的两个用处中,“用于舍入”与所采用的具体舍入方法有关,各种不同的舍入方法所需要的警戒位的位数是不同的,这在上一节中已经分析过了。下面重点分析“用于左规格化时移入尾数有效字长内”这一用处所需要的警戒位位数,最后再加上用于舍入所需要的警戒位就是警戒位的实际位数。

在警戒位的5种来源中,只有前3种(即加、减、乘、除运算)可能要进行左规格化,因此,在分析左规格化所需要的警戒位位数时只需考虑加、减、乘、除运算即可。

1. 加减法

如果两个同号的规格化浮点数相加或两个异号的规格化浮点数相减,则对阶时无论阶码较小的那个浮点数的尾数右移多少位,它与阶码较大的那个浮点数的尾数之和一定不需要进行左规格化。设 m_A 是阶码较大的那个浮点数的尾数, m_B 是阶码较小的那个浮点数对阶之后的尾数,在尾数采用原码、小数的情况下一定有:

$$\frac{1}{r_m} \leq |m_A| < 1, 0 \leq |m_B| < 1,$$

两尾数之和为:
$$\frac{1}{r_m} \leq |m_A + m_B| < 2,$$

$m_A + m_B$ 的和一定不需要进行左规格化,有可能要右规格化一位。

如果两个同号的规格化浮点数相减或两个异号的规格化浮点数相加,则根据两个浮点数阶差的不同会出现如下三种不同情况。针对尾数采用原码制的情况,举3个例子来说明浮点数阶差与左规格化位数之间的关系。

例 2.5 $0.10\ 001 \times 2^0 - 0.10\ 000 \times 2^0 = 0.10\ 000 \times 2^{-1}$

$$\begin{array}{rcl}
 & 0.10\ 001 & \times 2^0 \\
 \text{对阶:} & +1.10\ 000 & \times 2^0 \\
 \hline
 \text{求和:} & 0.00\ 001 & \times 2^0 \\
 \text{左规:} & 0.10\ 000 & \times 2^{-4}
 \end{array}$$

当两个规格化浮点数的阶差为 0 时,这两个浮点数的尾数之和最多可能要左规格化 $p-1$ 位(p 是尾数数值部分的字长),但是对阶时没有从尾数最低位移出任何代码,即对警戒位有需要但没有来源,因此不必设置警戒位。

例 2.6 $0.10\ 000 \times 2^0 - 0.11\ 111 \times 2^{-1} = 0.10\ 000 \times 2^{-5}$

$$\begin{array}{rcl}
 & p=5 & g=1 \\
 & 0.10\ 000 & 0 \times 2^0 \\
 \text{对阶:} & +1.10\ 000 & 1 \times 2^0 \\
 \hline
 \text{求和:} & 0.00\ 000 & 1 \times 2^0 \\
 \text{左规:} & 0.10\ 000 & 0 \times 2^{-4}
 \end{array}$$

当两个规格化浮点数的阶差为 1 时,这两个浮点数的尾数之和最多可能要左规格化 p 位,但是对阶时只从阶码较小的那个浮点数的尾数最低位移出了 1 位代码,因此也只需要设置一位警戒位。

例 2.7 $0.10\ 000 \times 2^0 - 0.11\ 111 \times 2^{-2} = 0.10\ 000 \times 2^{-1}$

$$\begin{array}{rcl}
 & p=5 & g=2 \\
 & 0.10\ 000 & 00 \times 2^0 \\
 \text{对阶:} & +1.11\ 000 & 01 \times 2^0 \\
 \hline
 \text{求和:} & 0.01\ 000 & 01 \times 2^0 \\
 \text{左规:} & 0.10\ 000 & 10 \times 2^{-1} \\
 \text{舍入:} & 0.10\ 001 & \times 2^{-1}
 \end{array}$$

如果两个规格化浮点数的阶差大于等于 2,对阶时可能要从阶码比较小的那个尾数的最低位移出去很多位,但是两个浮点数的尾数之和最多只可能左规格化 1 位,因此只需要为左规格化设置一位警戒位即可,另外保存下来的代码可以作为舍入用。

综上所述,无论什么情况下两个规格化浮点数相加,用于左规格化的浮点数位数只需要一位。

2. 乘法

两个规格化浮点数相乘,在尾数采用原码、小数表示的情况下,两个规格化尾数 m_A 、 m_B 及尾数乘积 $m_A \times m_B$ 的取值范围如下:

$$\begin{aligned}
 \frac{1}{r_m} &\leq |m_A| < 1, \quad \frac{1}{r_m} \leq |m_B| < 1, \\
 \frac{1}{r_m^2} &\leq |m_A \times m_B| < 1,
 \end{aligned}$$

两个规格化尾数的乘积 $m_A \times m_B$ 最多只需要左规格化一位,因此两个规格化浮点数相乘

时只需要为左规格化设置一个警戒位。

3. 除法

两个规格化浮点数相除,采用与乘法相同的分析方法,在尾数采用原码、小数表示的情况下,两个规格化尾数 m_A 、 m_B 及尾数商 $\frac{m_A}{m_B}$ 的取值范围如下:

$$\frac{1}{r_m} \leq |m_A| < 1, \frac{1}{r_m} \leq |m_B| < 1,$$

$$\frac{1}{r_m} \leq \left| \frac{m_A}{m_B} \right| < r_m,$$

两个规格化尾数的商 $\frac{m_A}{m_B}$ 不可能有左规格化,只可能发生右规格化一位或不需要规格化两种情况,因此两个规格化浮点数相除时不必为左规格化设置警戒位。

综合上面分析的几种情况及上节介绍的舍入方法,各种情况下所需要的警戒位位数如表 2.12 所示。

表 2.12 各种情况下所需要的警戒位位数

用于左规		加减法 1 位	乘法 1 位	除法 0 位	右规 0 位	十化二 0 位	总 计
用于舍入							
恒舍法	0 位	1	1	0	0	0	1 位
恒置法	-1 位	0	0	-1	-1	-1	0 位
下舍上入法	1 位	2	2	1	1	1	2 位
查表法	1 位	2	2	1	1	1	2 位
R * 舍入法	2 位	3	3	2	2	2	3 位

表 2.12 中第一行给出的是各种情况下用于左规格化所需要的警戒位位数,第一列给出各种舍入方法所需要的警戒位位数,各个交叉点上的数字表示各种情况下所需要的总的警戒位位数(包括用于舍入和用于左规格化两种用处),表的最后一列给出采用某种舍入方法时应该设置的警戒位位数。例如,采用下舍上入法一共需要设置两个警戒位,警戒位的高位用于在做加减法或做乘法时左规格化移入运算结果尾数的最低位,警戒位的低位用于舍入,如果在做加减法或做乘法时不需要进行左规格化,那么警戒位的高位用于舍入,它的低位无用。

另外,还应当指出,当浮点数尾数的基值大于 2 时,用于舍入的最低一位只需要设置一个二进制位。

也有少数大型和巨型计算机(如 CDC-6600)在中央处理机的通用寄存器中专门设置有警戒位寄存器(guard digit registers),这样可以多保存一些在运算过程中产生的警戒位,只是在必须送入内存存储器时才舍入到规定的字长,这是减少运算误差的一种好办法。还可以设置专门的舍入指令,针对不同的问题进行不同的舍入处理,从而为程序员能够直接控制运算误差提供了可能。

2.1.3 自定义数据表示

目前在大多数计算机中,数据存储单元(如寄存器、主存储器、外存储器等)里存放的都是纯数据,而对这些数据的类型(如定点数、浮点数、复数、字符、字符串、逻辑数、向量等)、进位制(如二进制、十进制、十六进制等)、字长(如字、半字、双字、字节等)、寻址方式(直接寻址、间接寻址、相对寻址、寄存器寻址等)、功能(如地址、数值、控制字、标志等)等的解释要通过指令中的操作码来进行。如 IBM 370 系列机中,仅算术加法指令就有 8 条,这些指令所用到的操作数的类型、字长、进位制和操作数所采用的寻址方式等均不相同,参见表 2.13。

表 2.13 IBM370 系列机中的加法指令

指令助记符	数据类型	字长(位)	进位制	寻址方式
AR	定点数	32	2	R-R
ADR	浮点数	64	尾数 16,阶码 2	R-R
AER	浮点数	32	尾数 16,阶码 2	R-R
AH	定点数	16	2	R-X
A	定点数	32	2	R-X
AD	浮点数	64	尾数 16,阶码 2	R-X
AE	浮点数	32	尾数 16,阶码 2	R-X
AP	定点十进制	64	10	S-S

寻址方式中的 R-R 表示参加运算的两个数都直接来源于通用寄存器,运算结果也送回通用寄存器,R-X 表示参加运算的一个数来源于通用寄存器,另一个数来源于主存储器,而且采用变址寻址方式,运算结果送回通用寄存器,S-S 表示参加运算的两个数都来源于主存储器,而且采用直接寻址方式,运算结果也送回主存储器。

然而,在人们比较习惯的高级语言和其它许多软件中,加法运算操作通常只有一个,而数据的属性必须在数据被引用前给以定义,如: $A = A + B$,编译器要根据前面定义的 A 和 B 的数据类型、字长、进位制、寻址方式等生成不同的加法机器指令。

例如,在 C 语言中常用的基本数据类型有:

int	基本整型,即定点数,长度为机器字长;
short	短整型,长度不长于一倍机器字长;
long	长整型,长度不短于一倍机器字长;
unsigned int	无符号整型;
unsigned short	无符号短整型;
unsigned long	无符号长整型;
float	基本实型,即短浮点数,在大多数机器中的字长为 32 位;
double	双精度实型,即长浮点数,在大多数机器中的字长为 64 位;

char 字符型。

从这里可以看到,在高级语言与机器语言之间存在着很大的语义差距,这种语义差距通常要靠编译器等系统软件来填补。

那么,很自然就产生了一个问题,能否在机器语言一级由数据自己定义其属性,从而简化指令系统,简化编译器。

早在 60 年代初期,美国的 Burroughs 公司就在一些大型计算机中引入自定义数据表示方式和带标志符的数据表示方式,下面,分别介绍这两种数据表示方式。

2.1.3.1 带标志符的数据表示法

美国 Burroughs 公司在 60 年代初期生产的 B5000 大型计算机中,每个数据有一位用来区分是操作数还是描述符;在 60 年代后期生产的 B6500 和 B7500 大型机中,每个数据用三位标志符来区分 8 种数据类型,如图 2.7 所示。在 70 年代生产的 R-2 试验性计算机中采用了 10 位标志符,如图 2.8 所示。



图 2.7 带标志符的数据表示方式

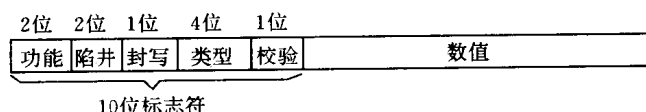


图 2.8 R-2 计算机中带标志符的数据表示方式

图 2.8 中的最高两位用来区分是操作数、指令、地址、控制字;两个陷阱位可由软件定义四种捕捉方式,为程序员对程序的进行跟踪控制提供方便;一个封写位指定数据是只读的还是可读可写的;四个类型位可以在最前面两个功能位定义的基础上进一步定义数据的类型,例如,如果最前面两位已经定义了是操作数,则四个类型位可进一步定义是二进制、十进制、定点数、浮点数、复数、字符串、单精度、双精度等,如果最前面两位已经定义了地址,则四个类型位可进一步定义是绝对地址、相对地址、变址地址、未连接的地址等;最后一个奇偶校验位。

从上面已经看到,标志符不仅可以用来定义数据类型,还可以用来描述机器中用到的各种有用信息,这就为硬件设计和软件设计提供了一种十分有效的手段。

标志符通常由高级语言的编译器或其它系统软件来建立,对一般高级语言程序员和计算机用户是透明的。

在采用标志符数据表示方式的机器中,大家首先担心的一个问题是:每个数据都带有标志符必然要加长数据的字长,从而程序所占用的存储空间将会增加。事实上这种担心是不必要的,因为数据的总存储量虽然加大了,但由于简化了指令系统,指令只需指出操作种类,不需要指出数据类型等,每条指令的字长可以缩短。只要设计合理,整个程序(包括指令和数据)的总存储量反而有可能减少,即使增加也不会增加得太多。

在一般程序中,一个数据通常要被多次访问;也就是说,有多条指令可能要访问同一个数据单元。下面我们举一个具体的例子来说明数据带有标志符后,整个程序占用存储空间的情况。

假设有 X 和 Y 两种不同类型的处理机, X 处理机的数据不带标志符,其指令字长和数据字长均为 32 位。Y 处理机的数据带有标志符,每个数据的字长增加至 35 位,其中有 3 位是标志符,其指令字长减少至 30 位。如果一条指令平均要访问两个操作数(因为有许多指令,如加、减、乘、除等运算型指令要访问两个源操作数、一个目的操作数共三个操作数,有的指令,如移位、增 1、减 1 等指令要访问一个源操作数、一个目的操作数共两个操作数,也有的指令只访问一个操作数,有的指令不访问操作数),每个操作数平均要被访问 R 次。现有一个程序,它的指令条数为 I ,下面我们分别计算在这两种不同类型的处理机中程序所占用的存储空间。

X 处理机,程序所占用的存储空间总和为: $B_x = 32I + \frac{2 \times 32I}{R}$,

Y 处理机,程序所占用的存储空间总和为: $B_y = 30I + \frac{2 \times 35I}{R}$,

为了便于比较,我们计算 Y 处理机与 X 处理机的程序所占用存储空间的比值:

$$\frac{B_y}{B_x} = \frac{30I + \frac{2 \times 35I}{R}}{32I + \frac{2 \times 32I}{R}} = \frac{15R + 35}{16R + 32}$$

当 $R > 3$ 时,有 $\frac{B_y}{B_x} < 1$,在实际应用中经常是 $R > 10$,即对于同样的程序,在 Y 处理机中所占用的存储空间比在 X 处理机中所占用的存储空间要小。

实际上,也可以用图 2.9 来说明这个问题,这个图表示在带标志符和不带标志符两种情况下程序所占用存储空间的比较。图中两个实线框分别表示在不采用标志符的情况下,指令和数据分别占用的存储空间的大小,并且假设这时的指令字长和数据字长是相等的。当数据带有标志符时,数据字长要加长,而指令字长将缩短。图中左上方阴影部分的面积表示因指令字长的缩短,程序减少的存储空间,左下方阴影部分的面积表示因数据字长加长,程序增加的存储空间。由于在一般程序中指令条数比数据条数要多,因此左上方阴影部分的高度必然大于左下方阴影部分的高度,因此,即使指令字长缩短的位数少于数据字长加长的位数,只要设计合理,很有可能使左上方阴影部分的面积大于左下方阴影部分的面积,从而,当数据带有标志符时,整个程序(包括指令和数据)所占用的总的存储空间很可能反而减少。

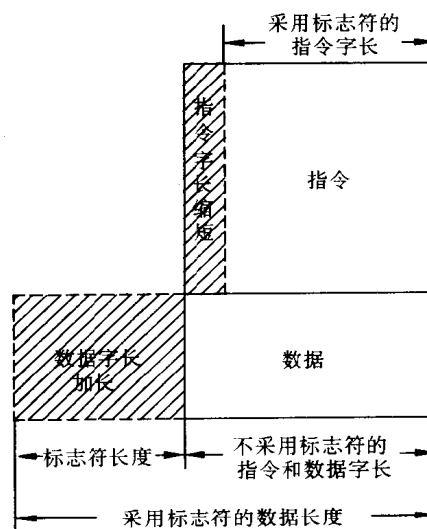


图 2.9 带标志符和不带标志符两种情况下程序所占存储空间比较

在采用标志符的数据表示法中,数据类型的一致性检查和转换(如一个浮点数加一个定点数,首先要把定点数转换成浮点数之后才能做加法运算)等都用硬件来实现,从而缩短了目标程序的长度,也就节省了程序所占用的存储空间。例如。在 IBM 370 系列机中执行 $A=A+B$ 运算,若 A 、 B 都是十进制数,只需要一条指令,共 6 个字节,在 IBM 370/145 机上的执行时间是 13 微秒,若 A 与 B 中有一个是定点二进制数,由于要进行数据类型的一致性检查和转换,在 PL/I 语言中的编译结果为 13 条指令,共 64 个字节,在 IBM 370/145 机上的执行时间是 408 微秒,两者相比,存储空间节省 5 倍,运算速度快 30 多倍。事实上,这种数据类型的一致性检查和转换在高级语言的优化编译器中经常要反复使用,有时在一个程序中就用成百上千次,如果每次都直接展开要占用大量的存储空间,如果改用子程序调用又会加长程序的运行时间。在采用标志符数据表示法的机器中,通常只要在控制存储器(ROM)中增加一小段微程序即可。

采用标志符数据表示方法主要优点有如下几个方面:

1. 简化了指令系统。指令中的操作码只需指出操作种类,不必指出数据类型,指令的条数必然减少很多,如本节一开始提到的在 IBM 370 系列机中仅加法指令就有 8 条以上,而在采用标志符数据表示法的机器中只需要一条加法指令。另外,如数据的进位制、字长、寻址方式、陷阱标志等都在数据中描述清楚了,不必再通过指令系统来进行描述,因而可简化指令的地址码,简化访问主存的指令、输入输出指令和程序控制指令。

2. 由硬件自动实现一致性检查和数据类型的转换。通过硬件能快速检测出参加运算的数据是否定义错误(如一个浮点数与一个字符相乘),是否不相容(如一个浮点数与一个定点数相加,两个字长不相等的定点数相加等)等。如果发现数据类型定义有错误,则进入出错中断处理程序;如果发现数据类型不相容,则由硬件自动进行数据类型和数据长度的变换。

3. 简化程序设计。数据带有标志符与人们的使用习惯更接近了,即缩小了人与机器之间的语义差距,从而使程序设计变得更加容易。通过硬件实现数据类型的一致性检查并自动进行数据类型的转换,减轻了程序设计人员的负担。

4. 简化编译器。每个操作数都通过自带的标志符来定义其数据类型、进位制、字长、寻址方式等,而指令中的操作码仅指出操作种类,这一做法与一般高级语言中先定义数据类型,然后才能使用的方法完全相同,从而使高级语言与机器语言之间的语义差距大大缩短了。另外,由于通过硬件自动实现数据的一致性检查和数据类型的转换,也简化了编译器的设计。

5. 支持数据库系统。一般的数据库系统都要求与数据类型无关,即一个软件不加修改就可适用于多种数据类型,标志符数据表示方式正好支持这种要求。在一般计算机系统中要实现软件与数据类型无关是非常困难的。

6. 方便软件调试。由于在每个数据中都有陷阱位,这些陷阱位可以通过软件来设置,也可以在调试程序时手工设置,这一功能为软件的跟踪和调试提供了极大的方便,必然会缩短软件开发的周期。

采用标志符数据表示方法的主要缺点是:

1. 数据和指令的长度可能不一致。从图 2.9 中可以看出,采用标志符数据表示方法

后,数据字长加长了,指令字长缩短了,因此,有可能造成数据字长与指令字长不相等。解决的办法通常有两个:一种解决办法是把数据和指令分别存放在两个存储器中。由于目前的计算机系统普遍存在主存储器的速度远远低于 CPU 速度的问题,采用多体存储方案是必然的趋势。然而,指令存储器与数据存储器的字长不同会给输入输出控制等造成麻烦,因为无论是指令还是数据,最终都要存放在外部的永久存储器(如磁盘存储器、磁带存储器等)上。另外,由于各种程序中数据和指令的比例可能差异很大,因而造成两种主存储器使用不平衡的问题。另一种比较好的解决办法是合理设计数据字长和指令字长,一般的设计原则是指令字长向数据字长靠拢。具体方法将在本章的“指令格式的优化设计”一节中作比较详细分析。

2. 指令的执行速度降低。由于每个数据都带有一个或几个标志符,在指令执行过程中要对每个标志符逐个进行解释,并判断数据是否相容。如果一条指令要用到几个数据,还要判断数据类型是否一致,如果不一致,还要进行数据类型的转换,因此,指令的执行时间必然会加长。然而,从前面的分析中我们已经注意到,采用标志符数据表示法会使程序的设计时间、编译时间和调试时间缩短。而一个程序的总开销通常是由设计时间、编译时间、调试时间和执行时间的总和决定的。前三项称为宏观速度,执行时间称为微观速度。采用标志符数据表示法提高了宏观速度,而降低了微观速度。将来,对计算机系统宏观速度的要求必然越来越高。从另一个角度看,在许多系统软件和应用软件中,数据的相容性、一致性,数据类型的转换等是由软件来实现的,这种做法与用硬件自动实现相比,其执行速度可能更低。

3. 硬件复杂度增加。由于要用硬件实现数据相容性、一致性测试,实现数据类型的自动转换,并解释所有标志符,因此采用标志符数据表示法的计算机系统,其硬件复杂度很大,过去仅在少数大型机和巨型机中采用。然而,随着 VLSI 技术的迅速发展,硬件复杂度将不成为一个大问题。美国 Intel 公司生产的 8087 微处理机,日本为研制第五代计算机而生产的个人计算机中都采用了标志符数据表示方法。

综上所述,采用标志符数据表示法虽然硬件复杂度高,机器的微观速度可能降低,但是,由于它能缩短人与机器之间的语义差距,能减少高级语言与机器语言之间的语义差距,能提高机器的宏观性能,因此这是一种很有前途的数据表示方法。目前有待研究的关键问题是如何定义好标志符,如何进一步扩大标志符的用途等。

2.1.3.2 数据描述符表示法

对于复杂的数据结构,如向量、矩阵、多维数组、记录等,其中有许多连续存放的数据的属性都是相同的,没有必要让每个数据都带有标志符。因此,在表示多维、或结构比较复杂的数据时要使用数据描述符。

数据描述符与标志符的主要区别是:标志符通常只作用于一个数据,而数据描述符要作用于的一组数据。因此,标志符通常与数值一起存放在同一个数据单元中,而数据描述符一般单独存放,独立占据一个存储单元。描述符用来描述一组数据的属性,包括整个数据块的访问地址、长度及其它特征或信息等。

图 2.10 给出的是在美国 Burroughs 公司生产的 B-6700 机中采用的一种数据描述符

表示方法。当最高三位为 101 时表示该字是一个数据描述符,最高三位为 000 时表示该字是一个数据。

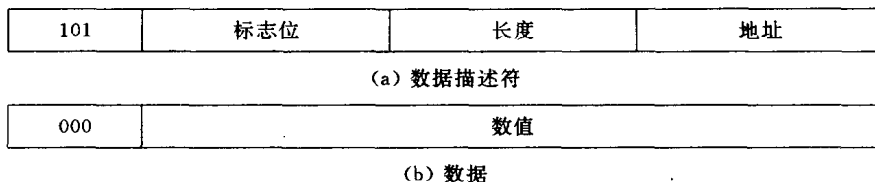


图 2.10 B-6700 机的数据描述符表示方法

图 2.10(a)中的标志位长度共 8 位,分别表示是否描述的是另一个数据描述符,描述的是单个数据还是数据块,是连续存放还是分段存放,是字还是串,是单精度数据还是双精度数据,是可读可写还是只读。如果被描述的是数据块,则“长度”字段指出这个数据块所包含的数据个数,“地址”字段指出这个数据块的首地址。

用数据描述符表示方法表示一个 3×4 矩阵 A 如图 2.11 所示。

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

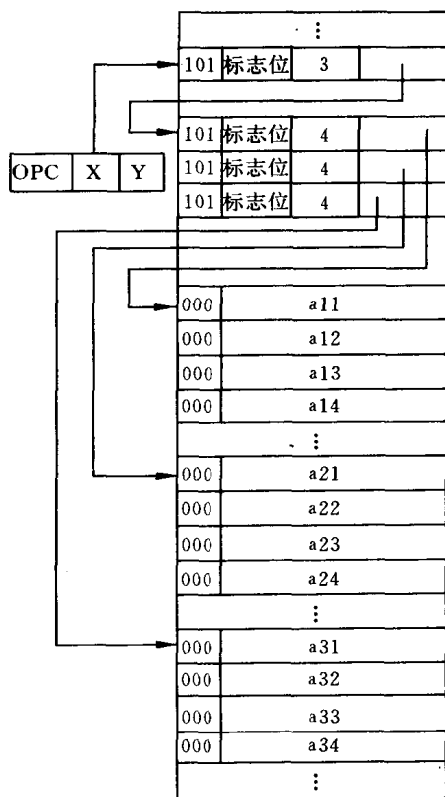


图 2.11 用数据描述符表示法
表示一个 3×4 矩阵

采用数据描述符可以表示多种结构不同的数据,把几个数据描述符按树形连接起来就能描述一个多维数组。例如,把一个两层的描述符按树型连接就能表示一个二维数组,具体描述方法可参见图 2.11。

一条指令中的一个地址 X 指向了一个描述符,这个描述符的“标志位”、“长度”和“地址”共同描述一个由三个描述符组成的描述符组,而组成这个描述符组的三个描述符又分别描述了三个数据块,每个数据块都由 4 个数据组成。采用两重树形结构的描述符能描述一个二维数组,同样,采用三重树形结构就能描述一个三维数组。

数据描述符方法为表示向量、矩阵和 multidimensional 数组等数据结构提供了一种很好的方法,能够用一条指令实现整个向量、矩阵或多维数组的运算。从图 2.11 中可以看到,采用数据描述符方法形成数据地址的速度比常规机器中采用的变址寻址要快得多,这样,也简化了编译器的代码生成。

采用数据描述符表示法的优点与缺点与上一节介绍的标志符数据表示方法相同,它的机器结构可能比标志符数据表示方法更复杂。

2.2 寻址技术

寻址技术是指寻找数据及其它信息的地址的技术,它是软件与硬件的一个主要分界面,是计算机系统结构的一个重要组成部分。

寻址技术要研究的主要内容包括编址方式、寻址方式和定位方式等。寻址技术研究的主要对象主要有寄存器、主存储器、堆栈和输入输出设备等,其中以面向主存储器的寻址技术为主要研究对象。

在《汇编语言程序设计》课中,我们已经学习了常见寻址方式的基本工作原理和它们的使用方法,在《计算机组成原理》课中,已经学习了寻址技术的实现方法等。本课程的重点是:在分析各种寻址技术优缺点的基础上,学习在计算机系统设计中如何选择和确定采用哪种寻址技术。

2.2.1 编址方式

要对寄存器、主存储器和输入输出设备等进行寻址,首先必须对这些设备进行编址。正像一个大楼里有许多房间,首先必须对每一个房间编上一个唯一的号码,人们才能找到要找的房间一样。

在计算机系统中,编址方式是指对各种存储设备进行编码的方法。主要包括编址的单位、零地址空间的个数等,另外还包括并行存储器的编址技术和输入输出设备的非线性编址技术等。

2.2.1.1 编址单位

目前常用的编址单位有字编址、字节编址和位编址等几种。以下,我们主要以主存储器的编址方式为例说明各种编址单位的优缺点。

字编址是实现起来最容易的一种编址方式,这是因为每个编址单位与设备的访问单位相一致,即每个编址单位所包含的信息量(如二进制位数)与访问一次设备(指读或写一次寄存器、主存储器和输入输出设备等)所获得的信息量是相同的。早期的大多数机器都采用这种编址方式,目前,仍有许多机器采用字编址方式。

在采用字编址的机器中,每取完一条指令,程序计数器加1,每从主存储器里读完一个数据,地址计数器加1,这种控制方式实现起来很简单,地址信息、存储器容量等没有任何浪费。它的主要缺点是没有对非数值计算的提供支持。从目前计算机的实际应用领域看,非数值应用已经超过了数值应用,而非数值应用要求按字节编址,因为它的基本寻址单位是字节。

在采用字编址的计算机中,需要设置专门的字节操作指令、位操作指令等,在这些指令中要有专门的字段指出操作数的字节编号或位的编号。

目前使用最普遍的编址单位是字节编址,这是为了适应非数值计算的需要,字节编址方式能够使编址单位与信息的基本单位(一个字节)相一致,这是它的最大优点。然而,如果主存储器的访问字长也是一个字节的话,那么主存储器的频带就太窄了,必将成为整个

计算机系统的瓶颈。通常主存储器的字长是一个字节的 4 倍以上,有的达到几十倍。由于编址字长与存储器的访问字长不一致,即每个编址单位所包含的信息量(一个字节)与访问一次存储器所获得的信息量(通常是一个字)不相同,从而就产生了数据如何在存储器里存放的问题。

图 2.12 给出了主存储器采用字节编址情况下,数据在主存储器中的三种不同存放方法,这里,存储器本身的字长都是 64 位(8 个字节),即一个存储周期能够从存储器中读或写一个字长为 64 位的数据,图中最左边一列是地址,用 16 进制表示,由于存储器的字长是 8 个字节,因此地址的最低 3 个二进制位是 000。

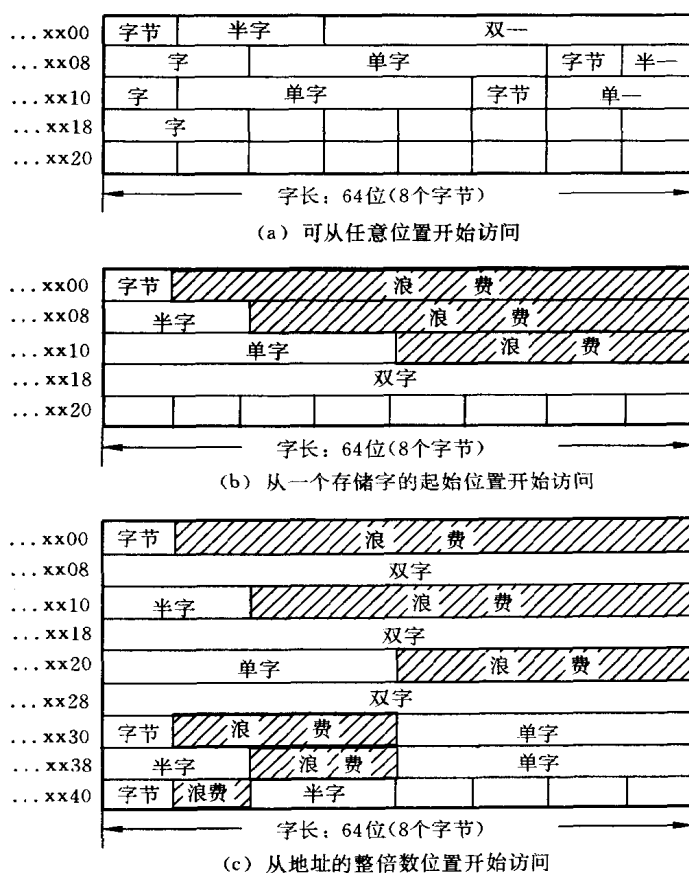


图 2.12 字节编址的主存储器的各种访问方法

图 2.12(a)是一种不浪费存储器资源的存储方式。可能有的四种不同长度的数据一个紧接着上一个存放。这种数据存放方式的优点是不浪费宝贵的主存储器资源,存在的主要问题有两个。一是除了访问一个字节之外,当要访问一个双字(在有些机器中把一个存储字的长度 64 位称为字,我们这里暂且把一个存储字称为双字,下同)、一个单字或一个半字时都有可能需要花费两个存储周期的时间,因为从图 2.12(a)中可以看出,一个双字、一个单字或一个半字都有可能跨越两个存储单元,这使存储器的工作速度降低了一

倍。另一个问题是存储器的读写控制比较复杂,图 2.13(a)表示当从存储器中读一个字节时,首先要用除掉最低 3 位之外的其余地址去读主存储器,实际上从存储器中读出了 8 个字节(因为存储器的访问字长是 64 位),然后用地址的最低 3 位控制一套多路开关从这读出的 8 个字节中选取一个字节输出。当要向主存储器中写一个字节时,操作起来就更复杂。如图 2.13(b)表示的那样,首先要把这个字节所在的整个双字从存储器中读出来,具体做法也是用除掉最低 3 位之外的其余地址去读主存储器,把读出的数据放在一个字长为 64 位的数据寄存器中,然后在地址的最低 3 位的控制下把要写入存储器的那个字节填入数据寄存器的相应字节中,数据寄存器的其余 7 个字节要保持不变,最后在原来给定的地址控制之下把数据寄存器中的整个双字写回主存储器。

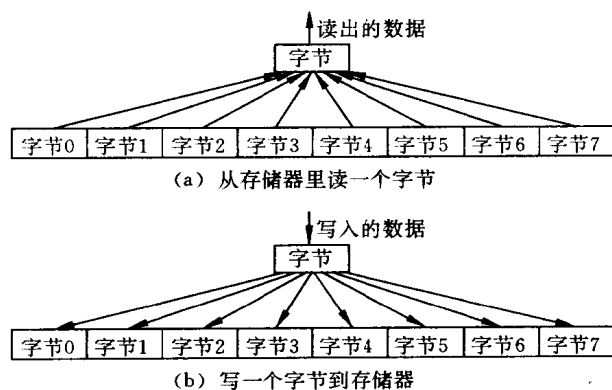


图 2.13 从存储器中访问一个字节的控制方法

这里应当特别指出的是,上述读出和写入共两次访问存储器操作实际上只需要一个存储器周期就能够完成,这是因为目前的绝大多数主存储器都采用动态存储器 DRAM 实现的,DRAM 是一种破坏性读出的存储器,每当从一个存储单元中读出数据之后,这个存储单元就被清除为 0。为了下次还能够从这个存储单元中读出原来的数据,必须把刚刚读出的那个数据重新写入这个存储单元,这一过程称为重写。向一个 DRAM 存储器中写入一个数据的过程也必须包括读出和重写两次访问存储器操作。所以,上述向主存储器中写入一个字节的读出和写入两次访问操作实际上就是利用 DRAM 存储器的读出和重写完成的,只需要一个存储器周期。

另外,在采用字节编址的计算机系统中,如果指令字长是 32 位,那么每执行完一条指令,指令计数器要加 4。如果像图 2.12 那样,存储器的字长是 64 位,当连续访问存储器时,每访问完一次,地址寄存器要加 8。

为了克服上述两个缺点,出现了如图 2.12(b)所示的另一种数据存储方式。这种存储方式规定,无论要存储的是字节、半字、单字或双字,都必须从一个存储字(在图中是 64 位)的起始位置开始存放(在图中是从最左边放起),而一个存储字的其余部分浪费不用。这种数据存储方式优点与缺点正好与图 2.12(a)中的存储方式相反,它的优点是:无论访问一个字节、一个半字、一个单字或一个双字都可以在一个存储周期内完成,读写数据的控制也比图 2.12(a)中的方式要简单。它的主要缺点是浪费了宝贵的存储器资源,如果双

字、单字、半字、字节四种不同字长的数据出现的概率相同的话,那么主存储器的实际利用率只有 50%,即有一半的存储器是浪费的。

综合图 2.12(a)和图 2.12(b)两种数据存储方式的优缺点,出现了如图 2.12(c)所示的折中方案。

实际上,具体分析上面所讲到的三个有关问题,存储器的访问时间是最主要的。像图 2.12(a)那样,访问一个数据可能需要两个存储器周期的存储方式,只有在对主存储器的速度要求不高,并要求充分利用存储器资源的应用场合才是适用的,在一般计算机系统中,主存储器的速度降低一倍是不能接受的。数据的读写控制复杂是最次要的,只要增加一些逻辑电路就能实现。存储器资源的浪费也很重要,但是在不得已的情况下,浪费一部分也是允许的。

图 2.12(c)所示的存储方式规定,双字地址的最末三个二进制位必须为 000,单字地址的最末两位必须为 00,半字地址的最末一位必须为 0。这种存储方式能够保证无论是访问双字、单字、半字或字节,都能在一个存储周期内完成,这是这种存储方式的最主要优点。尽管存储器资源仍然有浪费,但是浪费的资源比图 2.12(b)所示的存储方式要好得多。另外,这种存储方式的读写控制逻辑仍然比较复杂。

目前,在采用字节编址的主存储器中,图 2.12(c)所示的存储方式是使用最普遍的一种方法。

图 2.12 所示的三种存储方式都存在地址信息的浪费问题,例如,访问一个双字时实际使用了 8 个地址编码,当访问一个字时实际使用了 4 个地址编码,当访问一个半字时实际使用了两个地址编码,这是按字节编址必然要造成的问题。

另外,在采用字节编址方式的存储器中,一个存储字中的多个字节如何编码有两种方法。在图 2.14(a)中,从左边开始编址,这是软件设计人员的习惯,书写地址最好与书写普通数字一样从左向右。图 2.14(b)所示的是从右边开始编址,这是硬件设计人员的习惯,因为运算器的进位,程序计数器等进位都是从右向左进行的,但是,这种编址方式对软件设计人员使用起来就不太方便。例如,主存储器字长是 32 位(4 个字节),一个用 16 进制表示的 32 位数据 12345678H,在主存储器中的存储方式如图 2.15(a)所示,按地址从小到大的顺序从存储器中读出来的数据是 78563412H。一个单词 computer 在主存储器中的存储方式如图 2.15(b)所示,按地址从小到大的顺序从存储器中读出来的结果是 pmocretu,初学者使用起来往往很不习惯。在目前使用最广泛的 Intel x86 处理机中就采用这种存储方式。

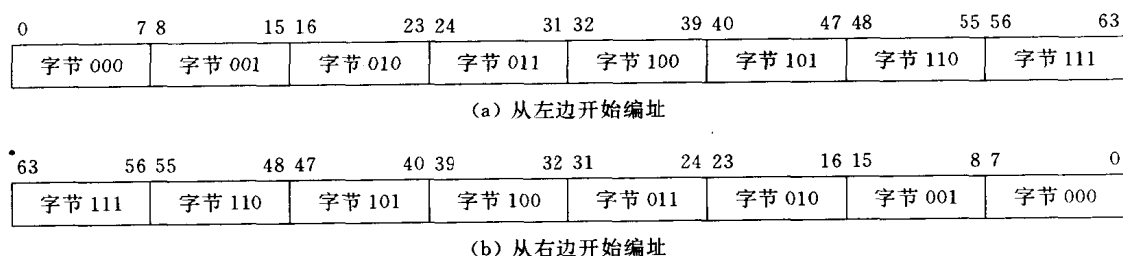


图 2.14 一个存储字的两种编址方式

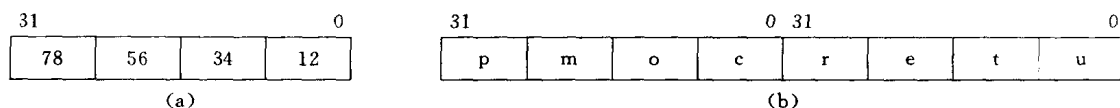


图 2.15 一种数据存储方式

也有部分计算机系统采用位编址方式,如 STAR-100 巨型计算机等。这种编址方式的主要优点和缺点与上面分析的字节编址方式相同,这里不再重述。所不同的只是地址信息的浪费更大。如果处理机的字长是 64 位,那么,一个字地址的最末 6 个二进制位为全 0,一个字节地址的最末 3 位为全 0,即访问一个字需要占用 64 个地址编码,访问一个字节需要占用 8 个地址编码。

目前处理机的字长已经逐渐趋向采用 64 位,而任何一个计算机系统的存储器,包括外存储器的存储容量离 2^{64} 还相差很远,所以地址信息存在的余量很大。另外,采用位编址方式对可变字长运算是有力的支持,如 PL/1 高级语言等。因此,位编址方式是一种很有应用前景的编址方式。

2.2.1.2 零地址空间个数

在计算机中需要编址的设备主要有运算器中的通用寄存器、主存储器和输入输出设备等三种。堆栈虽然也是一种存储设备,但是它不需要编址。还有一些特殊的寄存器,如变址寄存器、处理机状态字寄存器等,由于它们使用的场合很特殊性,这里不再介绍。

通用寄存器、主存储器、输入输出设备、堆栈等几种编址设备的工作速度、容量等性能差别很大,所要求的寻址方式和编址方式等也不相同,如表 2.14 所示。因此,对这几种存储设备的编址有如下几种:

1. 三个零地址空间。有许多机器对通用寄存器、主存储器、输入输出设备等三种存储设备分别进行编址。

表 2.14 几种编址设备的性能比较

存储设备	容量	工作速度	寻址种类	编址方式
通用寄存器	小	很快	单一直接寻址	单字或双字
主存储器	大	较慢	多种寻址方式	字、字节或位等
输入输出设备	较小	很慢	单一直接寻址	多种编址
堆栈	较小	较慢	隐含寻址	不编址

目前的 RISC(精简指令系统计算机 reduced instruction set computer)处理机通常有 32 个以上的通用寄存器,在有的处理机中,通用寄存器的数目已经达到几千个。运算器中所需要的原操作数主要来自于通用寄存器,运算结果也写回通用寄存器。

通用寄存器寻址技术的主要特点是:一般只有单一直接寻址方式,而且编址的长度与一次访问的长度通常相同,即采用简单的字编址方式。由于通用寄存器的存储容量比主存储器要小得多,因此所要求的地址码长度很短。

主存储器的寻址技术是最复杂的。由于存储容量大,因此所要求的地址码长度很长。寻址方式也比较复杂,必须有间接寻址或变址寻址等寻址方式。存储器的编址单位(通常是一个字节)与它的访问长度(通常是一个字)一般也不相同。另外,还有一个程序从浮动地址到物理地址的重定位问题。

输入输出设备的寻址技术与通用寄存器和主存储器的寻址技术都不相同。由于各种输入输出设备本身的性能差别很大,因此,所要求的编址方式也很不相同,例如,有的设备只需要一个地址码,而有的设备需要几个,甚至十几个地址码,各个被编址的信息的长度也不相同。另外,输入输出设备所要求的寻址方式通常比较单一,一般采用直接寻址方式就可以。

由于三种存储设备所采用的寻址技术差别比较大,因此对它们分别进行编址是很自然的。在许多机器中都采用三个零地址空间。

在采用三个零地址空间的计算机系统中,通常要有三类操作指令,包括三类数据传送指令、运算指令、移位指令和测试指令等。在许多 RISC 计算机中,为了简化指令系统,一般规定所有的运算、移位和测试等操作只能在通用寄存器中进行,只有访问操作在三类存储设备中都能进行。在许多 CISC 计算机中,通常所有操作都能在三类存储设备上进行,因此其指令系统比较复杂。

2. 两个零地址空间。通用寄存器独立编址,主存储器与输入输出设备统一编址。

一般在地址码的高端划出一部分来用作输入输出设备的地址,例如 4KB 左右。

主存储器与输入输出设备统一编址能够简化指令系统,因为在指令系统中不必另外设置输入输出指令,所有能够访问主存储器的指令都能够用来对输入输出进行操作,这是设置两个零地址空间的主要优点。

把输入输出设备与主存储器统一编址的主要缺点是指令执行过程复杂,执行时间要加长,因为所有访问主存储器的指令都要进行是否访问输入输出设备的判断,而且这种判断必须通过地址译码来进行。

对于 CSIC 处理机,除了有专门读、写主存储器的指令外,还有许多运算指令也能访问主存储器,包括一些功能复杂的运算指令。然而,访问输入输出设备的指令通常要求比较简单,其寻址方式一般为单的直接寻址方式,因此,对于输入输出设备,许多复杂的运算指令根本用不上。

对于 RISC 处理机,只有专门的 Load、Store 指令能够访问主存储器,一般的运算指令不能访问主存储器。所有运算操作都必须在寄存器中进行,即使是很简单的运算操作也要先把操作数读到寄存器中,然后在寄存器中进行运算,最后再把运算结果写回到主存储器中。而访问输入输出设备的指令通常只要求进行简单的操作,其寻址方式也是单的直接寻址方式,因此,在 RISC 处理机中,把输入输出设备与主存储器统一编址对于程序设计是不利的。

3. 一个零地址空间,即所有存储设备统一编址。地址的最低端是通用寄存器,最高端是输入输出设备,中间的绝大多数地址分配给主存储器。

采用一个零地址空间的处理机通常有一个存储容量比较大的高速存储器,有多个进程或线程存放在高速存储器中,并且可通过硬件进行进程和线程的切换。

有一些 DSP 芯片采用一个零地址空间的编址方式。

4. 隐含编址方式,实际上没有零地址空间。

在堆栈计算机中,运算指令是不需要地址的,有关设备(包括寄存器、主存储器和输入输出设备等)不需要进行编址。

Cache 存储器等采用联想方式访问的存储器是不需要编址的。

另外,在一般处理机中,一些特殊的寄存器或寄存器堆,如指令和数据的缓冲站等,是不需要编址的。

2.2.1.3 输入输出设备的非线性编址

计算机系统输入输出设备种类繁多,用途和性能也各不相同,因此,其接口上需要编址的寄存器的数目差别很大。例如,一台普通的打印机,其接口上有两个寄存器需要编址:一个是状态寄存器,另一个是数据寄存器。一台磁盘存储器的接口上有 5 个寄存器需要编址:磁盘地址寄存器、主存地址寄存器、交换个数计数器、控制与状态寄存器、数据寄存器等。

为了对各种各样输入输出设备的接口寄存器进行编址,目前的计算机系统有如下三种不同做法。

第一种:一台设备一个地址。实际上,这种编址方法仅对输入输出设备本身进行了编址,而没有对其接口上的寄存器进行编址。采用这种编址方法时,还必须通过指令中的操作码来识别该输入输出设备接口上的有关寄存器。通常要设置对数据寄存器进行读写的指令和设置对控制寄存器或状态寄存器进行操作的指令。有些机器还设置有对状态寄存器或控制寄存器中的某些位进行操作的指令,如启动设备或停止设备,测试设备是否忙,打开设备中断等。

一个设备一个地址的编址方法在早期的许多计算机中被广泛使用,目前在许多微型小型计算机中普遍使用。其主要优点是输入输出设备的地址很规整,地址码的长度很短。主要缺点是指令系统比较复杂。

第二种:一台设备两个地址。一个地址是数据寄存器,另一个地址是状态或控制寄存器。这种编址方式对于大多数输入输出设备恰到好处,因为绝大多数输入输出设备的接口上只有两个要编址的寄存器。这种编址方式与第一种编址方式相比,虽然设备地址扩大了一倍,但用于输入输出的指令系统简单了。目前,许多计算机系统的输入输出设备采用这种编址方式。

上述两种编址方式均存在有一个共同的问题:在设备接口上可能还有一些需要编址的寄存器没有分配地址。除了上述对设备接口上需要编址的寄存器采用直接编址的方式之外,还有两种编址方式也经常被采用。一种是依靠地址内部来区分,即设备的某一个地址对应有几个要编址的寄存器,这种编址方式仅适用于被编址的接口寄存器的长度比较短的情况。在不能依靠地址内部来区分的场合,也可以采用另一种称为“下跟法”的隐含编址方式。例如,要对某台设备上的两个寄存器进行操作,而只分配有一个地址,这时,我们可以对这两个寄存器事先隐含规定一个先后次序,那么奇数次使用这个地址的指令,对应的是设备接口上的第一个寄存器,偶数次使用这个地址的指令,对应的是设备接口上的第

二个寄存器。好比一个大人带着他的孩子去电影院看电影,两人可以使用一张票(孩子免票),电影院检票者通常认为,紧跟在成人后面的肯定是前者的孩子。

第三种:一台设备多个地址。根据各种输入输出设备的不同需要为它分配不同数目的地址。这种编址方式主要用于主存储器与输入输出设备合用一个零地址空间的计算机系统中,因为在这种计算机系统中可以分配给输入输出设备的地址数目相对比较多,例如,有些微型小型计算机系统把最高端的 4KB 或 8KB 的地址空间分配给输入输出设备。

2.2.1.4 并行存储器的编址技术

目前,一个主存储器通常都由多个完全独立的存储模块组成。如何对这多个存储模块进行编址有如下两种方法:

1. 地址码高位交叉编址

高位交叉编址方式的主要目的是用来扩大存储器容量。地址码的低位部分是各个存储体的体内地址,高位部分经过译码器译码后,用来区分存储体的体号。

目前,市场上可以买到 4MB、8MB、16MB、32MB、...的主存储器模块,用两个 8MB 的模块可以构成一个 16MB 的主存储器,用 4 个 8MB 的模块可以构成一个 32MB 的主存储器等。用户可以根据不同的需要随时很方便地改变自己的主存储器容量。这种模块化的主存储器通常都是采用高位交叉编址方法构成的。

采用高位交叉编址方法时,要求每个存储模块都有各自独立的控制部件,包括地址寄存器、地址译码器、驱动电路、放大电路、读写控制电路、数据寄存器等,在有些存储器模块中还有校验及校正电路、高速缓冲存储部件等。

主存储器采用高位交叉编址的方法,其主要优点是模块化结构好,用户可以很方便地扩充自己的主存储器。其缺点主要是,与单一模块的主存储器系统相比,控制部件的数量增加了。

2. 地址码低位交叉编址

低位交叉编址的主要目的是提高存储器速度。当然,在提高存储器速度的同时,由于增加了存储器模块的数量,同时也就增加了存储器的容量。

低位交叉编址方法地址码的使用方法与高位交叉编址方法正好相反,其低位部分是组成主存储器的各个存储体的体号,高位部分是各个存储体的体内地址。

为了达到提高主存储器速度的目的,在采用低位交叉编址的主存储器,在一个存储器周期内, n 个存储体必须分时启动,采用流水线方式工作。因此,低位交叉编址存储器实际上是一种采用流水线方式工作的并行存储器。在连续工作的情况下,保持每个存储体的速度不变,而整个主存储器的速度可望提高 n 倍。

目前的计算机系统一般都存在主存储器速度远远低于 CPU 速度的问题,在多处理机系统中这个问题尤为突出。除了采用多级高速 Cache 存储器之外,采用低位交叉编址的主存储器也是一种很好的方法。

采用低位交叉编址方法能够大幅度提高主存储器的速度,目前已经在共享主存储器的多处理机系统中得到广泛应用,许多高速的单处理机也采用了低位交叉编址的主存储器。其主要缺点是存在有访问冲突问题,使得由 n 个存储体组成的主存储器的加速比通常

要小于 n , 这一问题将在存储系统这一章中讨论。

2.2.2 寻址方式

指令在执行过程中通常需要操作数, 运算结果要送到存储单元中保持, 寻找操作数及数据存放单元的方法称为寻址方式。在《汇编语言程序设计》和《计算机组成原理》等课程中我们已经学习了寻址方式的基本原理, 本课程主要从计算机体系结构的角度介绍寻址方式的设计思想和设计方法。

2.2.2.1 寻址方式的设计思想

指令在执行过程中所需要的操作数(对目标地址是指运算结果所存放存储单元)通常有以下几种来源:

1. 直接在指令中给出操作数, 称为立即数寻址方式。这种寻址方式的优点是不需要数据存储单元, 指令的执行速度快。缺点是只能用于源操作数寻址, 数据的长度不能太长, 大量使用立即数寻址方式会使程序的通用性下降。立即数寻址方式通常仅仅用来指定一些精度要求不高的整型常数。

2. 寄存器寻址。指令在执行过程中所需要的操作数来源于寄存器, 运算结果也写回到寄存器中。这种寻址方式在所有的 RISC 计算机及大部分的 CISC 计算机中得到广泛应用, 因为目前的处理机中通常都有几十个、几百个甚至几千个寄存器。寄存器寻址方式的指令格式主要有:

```

OPC    R
OPC    R, R
OPC    R, R, R
OPC    R, M

```

前 3 种分别是一地址、二地址和三地址指令, 三地址指令主要用于向量计算机、VLIW(超长指令字)计算机中。第 4 种, 在 RISC 计算机中只允许 LOAD 和 STORE 指令使用这种形式, 而在 CISC 计算机中, 一般的运算指令也可以使用这种形式。

对于输入输出指令和一些特殊的处理机控制指令, 指令中所给出的寄存器可能是设备的控制寄存器、状态寄存器和处理机的程序计数器、堆栈指针、状态字寄存器等。

3. 主存寻址。这是几乎所有计算机中都必须采用的一类寻址方式, 其寻址种类也最为复杂。主存寻址的指令格式主要有:

```

OPC    M
OPC    M, M
OPC    M, M, M

```

主存寻址方式主要包括直接寻址方式、间接寻址方式和变址寻址方式等三种类型。

直接寻址方式是在指令中直接给出参加运算的操作数及运算结果所存放的主存地址, 即在指令中直接给出有效地址。在早期生产的计算机及目前的某些专用计算机中用得比较多。随着主存储器容量的不断扩大及虚拟存储器的普及, 这种寻址方式暴露出了许多

致命的弱点。首先,它需要很长的地址码,特别在二地址及三地址指令中,这一矛盾更为突出。通常的小型及微型计算机,表示一个主存地址要二进制 30 位左右,因此,有限长度的指令无法容纳如此长的地址码。其次,为了实现程序循环及有效地处理数组等运算,程序设计中修改数据的地址是必不可少的。采用直接寻址方式编写的程序,如果要修改数据地址就必须修改程序中的指令本身,采用这种方法编写的程序没有再入性,是现代程序设计思想所不能接受的。另外,目前使用的操作系统多为多任务或多用户的,采用直接寻址方式编写的程序会给操作系统的作业调度带来极大的不便,因为多任务及多用户操作系统要求程序能够在主存中浮动,所以,必须要有其它寻址方式来支持。

间接寻址方式在指令中给出的是操作数地址的地址,必须经过两次或两次以上的访问主存储器操作才能得到操作数。间接寻址可以只进行一次,也可以连续进行多次。多数计算机采用一次间接寻址方式,这种间接寻址方式只要用指令中给出的地址码去访问主存储器,就能直接得到操作数的有效地址。采用多级间址时,第一次间址的标志由指令给出,以后各次的间址标志要由紧接着访问主存储器所取出来的地址码给出,如果取出的地址码的间址标志位(通常指定最高位)为“1”,则要用除去标志位后的部分作为地址码继续访问主存储器,直至取出来的地址码的间址标志位为“0”时为止,这时,除去间址标志位之后的地址码即为有效地址。通常划出主存储器最低端的一小部分存放间接地址,因此,采用间接寻址方式时,指令中需要表示的地址码的长度可以很短。另外,也可以用寄存器来存放间接地址,这样,指令中需要表示的地址码的长度就更短。

采用变址寻址方式时,需要设置一个或多个变址寄存器。变址寄存器的长度由主存储器的寻址空间决定,例如,主存储器的寻址空间为 4GB,则变址寄存器的长度需要 32 位。也可以把某一个或几个通用寄存器兼作变址寄存器来使用。变址寄存器的主要作用是用来存放数组的基地址。

在指令中给出变址寄存器的编号(如果只有一个变址寄存器,可以隐含不表示)和地址的偏移量。当指令在执行时,用一个硬件的加法器,把变址寄存器中给出的基地址与指令中给出的地址偏移量作算术加法,相加的结果就是有效地址。

另外,变址寻址还有两种特殊的形式。一种是相对寻址方式,当变址寄存器就是程序计数器本身时称为相对寻址方式,采用相对寻址方式编写的程序本身就具有浮动性,称为与位置无关代码(PIC 码),这为程序的连接装配及在主存中的浮动调度提供了极大的方便。另一种是为了支持程序动态重定位的基址寻址方式,实际上,它也是一种变址寻址方式。在有些计算机系统(如 IBM 公司生产的大型中型计算机)中,既有变址寻址方式,又有基址寻址方式,这时,在指令执行过程中,对于每一个操作数都要进行两次变址运算,即做两次加法运算才能得到操作数的地址。

4. 堆栈寻址,堆栈寻址方式的地址是隐含的,在指令中不必给出操作数的地址,因此,指令的长度很短,一般的形式有:

OPC

OPC M

前一种是标准的采用堆栈寻址方式的指令,参加运算所需要的操作数从堆栈顶端弹

出,如果需要两个或多个操作数,则依次从堆栈顶端弹出,运算结果压入堆栈顶端。后一种堆栈指令在 RISC 计算机中仅仅用来在栈顶与其它主存储器单元之间交换数据,在 CISC 计算机中也可以用于运算指令,指令所需要的一个操作数来自于主存储器,其它操作数从栈顶弹出,运算结果压入栈顶。

2.2.2.2 间接寻址与变址寻址

对于数组运算,通常要用一个循环程序对数组中的各个元素进行操作,这时必须通过修改操作数的地址才能实现。间接寻址方式与变址寻址方式的设计目标都是为了解决操作数地址的修改问题。它们都能做到在程序设计过程中能够对操作数的地址进行修改,而不必去修改程序中的指令本身。

原则上,在一台计算机系统中,仅需设置间址寻址方式与变址寻址方式中的任何一种即可,如在 IBM 公司生产的大中型计算机系统中,只有变址寻址方式,没有间址寻址方式。在许多小型及微型计算机系统中,只有间址寻址方式,没有变址寻址方式。也有一些计算机系统,间址寻址方式和变址寻址方式两种都有。

那么,就会提出这样一个问题:在设计一台计算机系统时,如何选取间址寻址方式与变址寻址方式?下面,我们通过一个简单的例子,分析这两种寻址方式的区别及各自的优点及缺点。

例 2.8 一个由 N 个元素组成的数组,已经存放在主存储器的连续存储单元中,现要把它搬到主存储器的另一个连续的存储单元中,源数组的起始地址为 AS,目标数组的起始地址是 AD,不必考虑可能出现的存储单元的重叠问题。为了编程简单,采用一般的两地址指令编写程序。

首先,用间接寻址方式编写程序如下:

```
START:  MOVE    ASR,    ASI    ;保存源数组的起始地址,为了程序具有再入性
        MOVE    ADR,    ADI    ;保存目标数组的起始地址
        MOVE    NUM,    CNT    ;保存数据的个数
LOOP:   MOVE    @ASI,    @ADI    ;用间址寻址方式传送数据
        INC      ASI          ;源数组的地址增量
        INC      ADI          ;目标数组的地址增量
        DEC      CNT          ;个数减 1
        BGT     LOOP         ;测试 N 个数据是否传送完
        HALT                ;停机
ASR:    AS
ADR:    AD
NUM:    N
ASI:    0
ADI:    0
CNT:    0
```

;源数组的起始地址
;目标数组的起始地址
;需要传送的数据个数
;当前正在传送的源数组地址
;当前正在传送的源数组地址
;剩余数据的个数

为了程序具有再入性,前 3 条指令是必须的,数据存放单元要分别重复设置一份也是必须的。

然后,用变址寻址方式编写程序如下:

```
START:  MOVE    AS,    X           ;把源数组的起始地址送入变址寄存器
        MOVE    NUM,  CNT         ;保存数据个数,为了程序具有再入性
LOOP:   MOVE    (X),   AD-AS(X)   ;AD-AS 为地址偏移量,在汇编时计算
        INC     X              ;增量变址寄存器
        DEC     CNT           ;个数减 1
        BGT     LOOP          ;测试 N 个数据是否传送完成
        HALT                    ;停机
NUM:    N
CNT:    0                       ;需要传送的数据个数
                           ;剩余数据的个数
```

比较以上两个程序,可以很明显地看出,采用变址寻址方式编写的程序简单、易读。对程序员来说,两种寻址方式的主要差别如下:

间址寻址方式:间接地址在主存储器中,没有偏移量。

变址寻址方式:基地址在变址寄存器中,带有偏移量。

由此产生的两种寻址方式的优点与缺点是:

1. 实现的难易程度:间址寻址方式实现起来很容易,只需要增加一条从主存储器的数据寄存器到地址寄存器的数据通路即可。实现变址寻址方式需要增加较多的硬件,需要一个硬件的加法器,一个或多个变址寄存器(也可以与通用寄存器合用)。

2. 指令的执行速度:采用间址寻址方式编写的程序,执行速度很慢。读写一个操作数至少需要访问两次主存储器,第一次访问主存储器读取有效地址,第二次访问主存储器是读或写操作数,如果采用多次间址寻址方式,则访问主存储器的次数还要增多。例如执行上述程序中的 `MOVE @AS, @AD` 这条指令,至少需要访问主存储器 5 次,其中,第一次访问主存储器,读取指令本身,第二、第三次访问主存储器读到源操作数,再经过两次访问主存储器操作,才能把数据送入主存储器的目标地址单元中。采用变址寻址方式编写的程序,执行速度比较快。有效地址通过硬件加法器直接产生,不需要访问主存储器。如执行上述程序中的 `MOVE (X), AD-AS(X)` 这条指令,只需要访问主存储器 3 次。

3. 对数组运算的支持,变址寻址方式比较好,间址寻址方式较差,这是因为变址寻址方式可以带有偏移量。基地址加偏移量能够很有效地表示向量、矩阵等数据。

另外,对于变址寻址方式和间址寻址方式,还有以下几个问题需要注意:

1. 变址寻址方式中的偏移量是带有符号的,通常用补码表示,这样,不仅可以有向前的偏移,也可以有向后的偏移。偏移量的长度一般短于基地址的长度,在做加法时,一定要把偏移量的符号扩展,直至与基地址的长度相同。例如,某机器的偏移量为二进制八位,有一个用十六进制补码表示的偏移量 F3,基地址为二进制十六位,有一个用十六进制表示的基地址 05CD,正确的做法是:在做加法之前,首先要将偏移量的符号扩展 8 个二进制位,成为 FFFD,再与基地址相加,结果的有效地址是:05CD+FFFD=05C0。

2. 自动变址。从上面的例子中看到,无论采用间接寻址方式,还是采用变址寻址方式编写程序,对于数组运算,都必须有对地址进行增量的指令。在采用间接寻址方式编写的程序中,有两条指令分别对源数组和目标数组的地址指针进行增量,在采用变址寻址方式

编写的程序中,有一条指令对变址寄存器进行增量。为了省去这些对地址进行增量的指令,在许多机器中对于间接寻址方式和变址寻址方式都增加了自动变址的功能。

上例中,对于采用间接寻址方式编写的程序,只要把指令:MOVE @AS, @AD 改为: MOVE (AS)+, (AD)+,紧接在下面的两条分别对 AS 和 AD 指针作增量的指令就可以省去。同样,在采用变址寻址方式编写的程序中,把指令:MOVE (X), AD-AS (X) 改为: MOVE (X), AD-AS(X)+,紧接在下面的一条对变址寄存器作增量的指令就可以省去。

地址增量的单位,要根据具体机器所采用的编址方式和数据元素的长度等关系来确定。例如。对于采用字节编址的机器,如果数据元素的长度是二进制 16 位,则增量单位是 2,如果数据元素的长度是二进制 32 位,则增量单位是 4。

地址增量的先后关系也很有讲究,有如下几种方式:

第一种:先用后增与先减后用方式。程序中的写法是:(X)+,-(X)。这种方式多用于有后进先出堆栈,而且堆栈指针指向栈顶元素的机器中。

第二种:先增后用与先用后减方式。程序中的写法是:+(X),(X)-。这种方式多用于有后进先出堆栈,而且堆栈指针指向栈顶第一个空元素的机器中。

第三种:先增后用与先减后用方式。程序中的写法是:+(X),-(X)。这种方式多用于有没有后进先出堆栈的机器中。

3. 前变址与后变址。在既有变址寻址方式,又有间址寻址方式的计算机系统中,先做变址运算还是先做间址运算,需要事先定义。在指令中给出一个变址寄存器编号 X 和一个相对地址 A。有效地址 EA 的计算方法有如下两种:

第一种:前变址寻址方式。有效地址的计算过程是: $EA = ((X) + A)$ 。

第二种:后变址寻址方式。有效地址的计算过程是: $EA = (X) + (A)$ 。

2.2.2.3 寄存器寻址

在许多处理机中,特别是 RISC 处理机中,通常有 16 个,甚至数百个通用寄存器(或称为累加器、数据寄存器等),指令在执行过程中所需要的操作数直接取自通用寄存器,运算结果也保存在通用寄存器中,这样,运算型指令只要指定通用寄存器的编号,无须指定主存储器的地址,这就是寄存器寻址方式。

寄存器寻址方式也可以有间接寻址,即在通用寄存器中存放的是操作数在主存储器中的地址,或者是操作数在主存储器中的地址的地址等等。寄存器间接寻址方式与主存储器的间址寻址方式一样能够解决操作数地址的修改问题,可以做到在程序设计过程中对操作数所存放的主存地址进行修改(只要修改存放主存地址的通用寄存器中的内容即可),而不必去修改程序中的指令本身。

寄存器寻址方式的优点主要有:

1. 指令字长短。由于通用寄存器的数量一般只有几十个,在指令中只需很少几位就能表示一个操作数的地址。例如,在 IBM370 系列计算机中,有 16 个通用寄存器,只要用 4

个二进制位就能表示一个操作数的地址,即使是三地址指令,也只要 12 位地址码。

由于通用寄存器的字长可以比较长(多为 32 位、64 位等),在采用寄存器间址寻址方式时,在通用寄存器中可以存放字长很长的主存地址。对于数据字长为 64 位的计算机系统,在一个通用寄存器中不仅可以存放一个字长很长的主存地址,还可以同时存放地址的偏移量及各种标志等其它许多信息。

2. 指令执行速度快。由于寄存器的速度很快,与主存储器相比,访问时间几乎可以忽略不计,因此,大多数寄存器型指令都能在一个节拍内完成。

对于那些要连续使用的数据,把它们存放在通用寄存器中,能够大幅度提高程序的执行速度。这里要特别指出的是,对于用得最为普遍的三地址指令,必须要有通用寄存器的支持,否则,程序的执行速度将明显下降。关于这一点,在指令系统一节中还要作详细的分析。

3. 支持向量、矩阵运算。当通用寄存器的数量比较多时,可以把一个向量或向量的一部分放在通用寄存器内,从而提高运算速度。

寄存器寻址方式也有明显的缺点,主要有:

1. 不利于优化编译。由于通用寄存器的速度与主存储器相比要快得多,因此,通用寄存器分配得是否合理,直接影响到程序的执行速度。通常要把那些连续使用或用得比较频繁的变量分配在通用寄存器中,这就给编译器的优化设计造成了很大的困难。这说明,通用寄存器型机器不适应高级语言的数据模型。

另外,通用寄存器的不对称性也给优化编译带来很大的麻烦。例如,在 IBM370 系列机中规定:R0 寄存器不能作变址寄存器和基址寄存器使用,对于双字长指令只能用编号是偶数的通用寄存器等。在小型计算机 PDP-11 中,限制就更多:R0 用于宏调用,隐含存放返回值,R5 在高级语言与汇编语言连接时使用,R6 兼作堆栈指针,R7 兼作程序计数器,R0 至 R5 可兼作变址寄存器,对于双字长指令也只能用编号是偶数的寄存器等。

2. 现场切换困难。在程序运行过程中,当发生调用、中断、分时切换等情况时,要把有关通用寄存器中的内容都保存到主存储器中,在程序返回时,再全部恢复这些通用寄存器中的内容。通用寄存器的数量越多,保存和恢复所需要的时间就越长。

为了解决这一问题,目前的多数处理机都设置有两套或两套以上的通常寄存器,程序员只能看到其中的一套通用寄存器,当发生现场切换时,硬件自动切换到另一套通用寄存器。例如,美国 SUN 公司的 SPARC 处理机,设置有 8 套通用寄存器,采用重叠寄存器窗口技术,程序员能够看到的 32 个通用寄存器分成三个部分:局部寄存器、与上层调用连接的寄存器和与下层调用连接的寄存器,在与上层调用连接的寄存器中存放由上层调用带给本层的变量,在与下层连接的寄存器中存放本层调用传送给下一层调用的变量,最后一层的与下层连接寄存器和第一层的与上层连接寄存器完全重叠。

3. 硬件复杂。一方面,在处理机中设置大量的通用寄存器,包括程序员看见的寄存器及更多的程序员看不见的寄存器,需要增加硬件,另一方面,这些寄存器的读、写及现场切换等的控制也相当复杂,例如,一个有 8 个功能部件的超标量处理机,在一个节拍中需要从通用寄存器中读出 16 个操作数,写回 8 个运算结果,可见其译码控制逻辑是相当复杂的。

2.2.2.4 堆栈寻址方式

从 60 年代开始,出现了一批以堆栈寻址方式为主的堆栈计算机。这类计算机系统与上面提到的以寄存器寻址方式和主存寻址方式为主的计算机系统相比,在一定程度上缩小了高级语言与机器语言的差距。堆栈计算机具有如下特点:

1. 支持高级语言,有利于编译程序。因为一般的算术表达式可以很容易地转化成逆波兰表达式,而逆波兰表达式能够直接形成由堆栈指令组成的程序,这样就简化了编译程序。在指令系统这一节中,还要举例分析。

以主存寻址方式为主的计算机系统,在编译一个算术表达式时,要为每一个变量分配主存单元,另外,还会人为地产生一些中间变量。如何减少中间变量的个数,合理地分配存储单元,是编译器的一项相当困难的工作。

对于以寄存器寻址方式为主的计算机系统,编译器需要决定哪些变量放在通用寄存器中,哪些变量放在主存中,以减少访问主存储器的次数。另外,也同样存在如何减少中间变量,节省存储空间的问题。

2. 程序的总存储量最短。由于堆栈指令不需要地址码,指令的长度很短,与以寄存器寻址方式和以主存寻址方式为主的计算机系统相比,虽然程序本身的条数没有减少,但程序的总存储量要缩短许多。

3. 支持程序的嵌套和递归调用,支持中断处理。嵌套调用是指一个子程序又调用另一个子程序,递归调用是指一个子程序直接或经过别的子程序间接调用它本身,由此又可分为直接递归调用和间接递归调用。

在程序调用过程中,要保存返回地址,保存处理机状态,保存程序现场,并向子程序传送参数。在堆栈型计算机中,可以把这些信息都压入堆栈,而不必为它们赋予地址。当从子程序返回时,可以直接从堆栈中弹出所需要的信息。这样,可以减少大量的辅助操作,加快运算速度。

中断的处理过程与程序的调用很类似,使用堆栈能够加速中断的处理过程,简化中断程序设计。

堆栈型计算机的主要缺点是运算速度比较低,这是由于堆栈与处理机之间的信息传送量很大造成的。

实际上,对堆栈访问最频繁的是堆栈顶部的几个单元。为了提高堆栈的工作速度,许多堆栈型计算机的栈顶部分设计成一个高速的寄存器堆。这样,访问堆栈就像访问寄存器一样快速。

目前,许多以寄存器寻址方式和主存寻址方式为主的计算机系统,也设置有堆栈,用以支持程序的嵌套和递归调用,支持中断处理。

2.2.3 定位方式

为了把一个程序交给处理机运行,必须先把这个程序的指令和数据装入到主存储器的一个或几个区域中。在一般情况下,程序所分配到的主存物理空间与程序本身的逻辑地址空间是不相同的,因此,必须要把指令和数据中的逻辑地址(相对地址)转换成主存储器

的物理地址(绝对地址),这一转换过程称为程序的定位。定位方式主要研究程序中的指令和数据的主存物理地址在什么时间确定?采用什么方式来实现?

程序需要定位的主要原因有如下几个:

1. 程序的独立性。随着计算机应用范围的扩大,要解决的问题越来越复杂。程序员希望摆脱麻烦的存储分配,而把主要精力用于研究算法。在高级语言及其它各种面向应用的符号语言研究出来之后,程序员只要用符号命令、数据说明及各种输入输出说明来编写程序,即完全用符号名称来访问信息。在这种情况下,当编译程序对源程序进行编译时,也不必考虑主存储器的实际地址,只要把目标模块安排在从零开始的地址空间中。以后,每当需要运行这个程序时,由操作系统根据当时主存储器的实际使用情况,决定程序的主存物理地址。

2. 程序的模块化设计。在设计一个大的应用程序时,通常要把它分解成几个相对独立的部分,各个部分分别独立地编写程序,并分别进行编译,这样,一个程序将由几个独立的模块组成,这些模块可以在程序执行之前,甚至推迟到程序执行过程中需要时才装配链接起来。在这种情况下,编译程序在对某段程序进行编译时,无法知道其它模块占用存储空间的情况,甚至不知道整个程序是由那几个模块组成的,只有在程序装入到主存时,或在实际运行时才能确定指令和数据的主存物理地址。

3. 许多应用问题涉及到表、队列、堆栈等数据结构,这些数据结构在程序运行过程中,其大小往往是变化的,因此,要根据程序的实际情况,动态分配主存储器的物理地址。

4. 有些程序本身很大,大于分配给它的主存物理空间,因此,要求当程序的一部分装入主存后就能够开始运行,并且在运行过程中,根据需要再装入程序的其它部分。这就要求系统能够在程序运行过程中,动态确定指令和数据的主存物理地址。

根据程序中指令和数据的主存物理地址的确定时间,定位方式可分为三种:直接定位、静态定位和动态定位。在程序装入主存储器之前,程序中的指令和数据的主存物理地址就已经确定了称为直接定位方式。在程序装入主存储器的过程中随即进行地址变换,确定指令和数据的主存物理地址的称为静态定位方式。在程序执行过程中,当访问到相应的指令或数据时才进行地址变换,确定指令和数据的主存物理地址的称为动态定位方式。

以下,首先简单说明程序逻辑地址及主存物理地址的基本概念,然后分别介绍上述这三种定位方式。

2.2.3.1 逻辑地址与物理地址

在高级语言、汇编语言等符号语言出现之后,程序员可以直接用符号指令、数据说明、输入输出操作说明来编写程序,这种程序称为源程序。在源程序中,用符号名称来表示地址,即程序员实际上是在一个符号名称空间内编写程序的,如图 2.16(a)所示。当编译程序对源程序进行编译时,将符号元素转换成由指令和数据组成的目标程序,并把符号地址转换成存储器的地址。

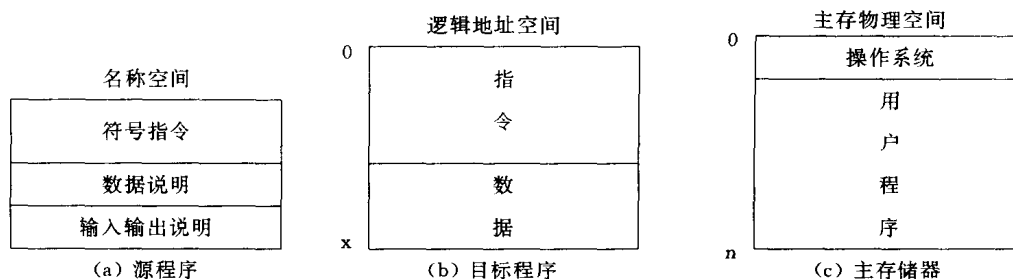


图 2.16 逻辑地址与主存物理地址

早期的计算机系统、目前的某些小型微型计算机和一些专用计算机，在编译源程序时，就已经可以确定程序在主存储器中将要存放的实际位置，因此，能够把指令和数据的地址直接编译成主存储器的物理地址。但是，对目前的多数计算机系统而言，编译程序在对一个源程序或源程序段进行编译时，还不能确定程序在主存储器中将要存放的实际位置，因此，只能各自为政，各个源程序或程序段都从自己的零地址开始分配地址空间，这种地址称为逻辑地址，或相对地址，如图 2.16(b)所示。

通常，在一台计算机系统中只有一个主存储器，是一维线性编址的，这个主存储器的地址称为物理地址，或绝对地址，如图 2.16(c)所示。

总之，逻辑地址是指相对于本程序或本程序段的相对地址，而物理地址是程序在主存储器中使用的实际地址。逻辑地址的集合称为逻辑地址空间，主存物理地址的集合称为主存物理空间。

2.2.3.2 直接定位方式

采用直接定位方式的前提条件是：程序员在编写程序时，或编译程序在对源程序进行编译时，就已经确切知道该程序应该占用的主存物理空间，因此，他们可以直接使用实际的主存物理地址来编写或编译程序。即使在多道程序环境下，也可以保证所使用的主存物理地址不相互重叠。

在单任务系统中，一个程序一旦开始运行，在其整个执行过程中，计算机的全部资源都为它所独占，因此，采用直接定位方式是可行的。

在多任务或多用户系统中，可以把整个主存物理空间划分为若干个固定的，大小相同的分区，并为每个任务分配相应的分区。因此，对程序员或编译程序而言，主存储器的可用物理空间是已知的，能够在程序装入主存之前确定指令和数据的物理地址。

如果程序比较大，其存储容量超过了分配给该程序的主存物理空间，这时，可以把整个程序分割成若干个既有联系，又相对独立的程序段，在程序运行过程中逐段调入相应的主存物理空间，这种技术称为“覆盖”。

2.2.3.3 静态定位方式

静态定位是由专门设计的定位装入程序来完成的，并要求程序本身是可以重定位的，即那些要修改地址的指令和数据要具有某种标识。静态定位要求程序在运行之前，在装入

主存储器的过程中集中一次完成地址变换,把那些带有标识的指令或数据中的逻辑地址全部变换成主存储器的物理地址。程序一旦进入主存储器之后,就不能再在主存储器中“搬家”了。

静态定位方式与直接定位方式所不同的是:静态定位方式允许程序每次运行时装入到不同的主存物理空间中,而直接定位方式由于程序在装入主存储器之前,所有指令和数据的地址都已经是唯一确定了的主存物理地址,因此,它只能装入到一个固定的主存物理空间中。图 2.17(a)表示程序 A 的第一种运行情况,它被装入到从 n 开始的主存物理空间中,如果该程序中有一条指令是 `JMP 100`,在程序装入到主存储器的过程中要把这条指令变换成 `JMP 100+n`。同样,图 2.17(b)表示程序 A 的第二种运行情况,它被装入到从 m 开始的主存物理空间中,这时,逻辑地址空间中的同一条指令 `JMP 100`,在程序装入到主存储器的过程中要把它变换成 `JMP 100+m`。

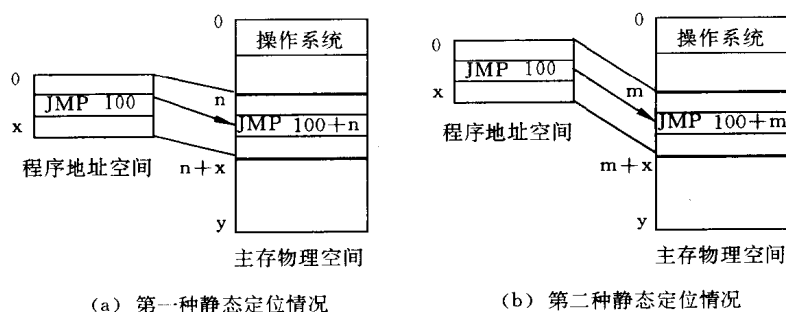


图 2.17 程序的静态定位方式

采用静态定位方式的主要优点是:

1. 不需要增加任何硬件设备,可以在一般机器上全部用软件实现。
2. 静态定位装入程序可以对由多个程序段组成的程序进行静态链接,而且实现起来比较简单。

静态定位方式主要缺点是:

1. 因为程序是在执行之前一次装入到主存储器中的,在执行期间程序不能在主存储器中移动,所以对提高主存储器的利用率不利。
2. 若程序所需要的存储容量超过了分配给它的主存物理空间,则程序员必须采用覆盖结构。
3. 多个用户不能共享已经存放在主存储器中的同一个程序,如果几个用户要使用同一个程序,则每个用户必须在各自的主存空间中存放一个程序副本。

2.2.3.4 动态定位方式

采用动态定位方式必须有硬件支持,它采用与变址寻址方式相同的方法,把程序的逻辑地址转换成主存的物理地址。

如图 2.18 所示,程序在装入主存储器时,指令和数据的地址不作任何修改,只把主存的起始地址存入与该程序相对应的基址寄存器中。在程序执行时,只要用地址加法器将指

令中的逻辑地址(在寻址方式中称有效地址)与已经存放在基址寄存器中的程序起始地址(在寻址方式中称基址)相加,就能形成主存的物理地址。实际上,并不是所有指令中的地址码都是要修改的,例如,那些采用相对寻址方式的指令地址码不需要修改,为此,必须在

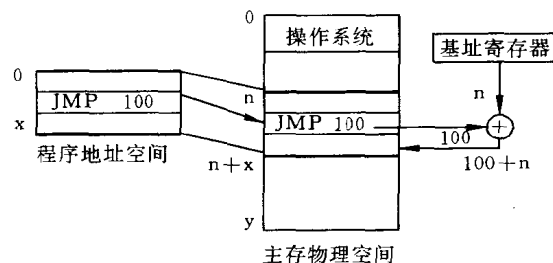


图 2.18 程序的动态定位方式

指令中给予指明,指出本指令中主存地址码是否需要加基址。

IBM-370 系列机的一种典型指令格式如下:

操作码	R1	X2	B2	D2
-----	----	----	----	----

R1 是第一操作数,在通用寄存器中,由 X2、B2、D2 共同形成第二操作数,其中,X2 是变址寄存器,B2 是基址寄存器,D2 是地址的偏移量。主存的物理地址经过两次加法(都用硬件加法器实现)得到:

$$\text{主存物理地址} = (X2) + (B2) + D2$$

变址寄存器、基址寄存器与通用寄存器合用,共有 16 个。当 X2=0000 时,表示不变址,当 B2=0000 时,不加基址。

虽然基址寻址与变址寻址硬件的实现方法完全相同,但它们在程序中的作用是根本不同的。变址寻址支持用循环程序对数组进行运算,而基址寻址支持把程序的逻辑地址变换成主存的物理地址,以实现程序的动态定位。它们的共同目标是不必修改程序中指令(为了程序具有再入性),而达到修改地址码的目的。

动态定位方式实质上是把静态定位方式中用链接装入程序实现的对指令地址的修改,改用硬件的地址加法器和基址寄存器来实现,由此带来许多好处。

采用动态定位方式的主要优点有:

1. 由于程序是在实际执行时,由硬件形成主存物理地址的,因此,在程序开始执行之前,不一定要把整个程序都调入到主存中,而且,一个程序可以被分配在多个不连续的主存物理空间内,从而,可以使用较小的主存分配单位,以提高主存储器的利用率。

2. 几个程序可以共享存放在主存中的同一个程序段,而不必在主存中存放多个副本,这是前两种定位方式无法实现的。

3. 支持虚拟存储器。可以为用户提供一个比实际主存储器的物理空间大得多的逻辑地址空间。对于大程序,不需要采用覆盖结构,程序的调度完全由系统管理程序来实现。

动态定位方式的主缺点有:

1. 需要有硬件支持。

2. 实现存储管理的软件算法比较复杂。

2.3 指令格式的优化设计

由于指令系统是程序设计人员所能看到的计算机的主要属性,它在很大程度上决定了整个计算机系统所具有的基本功能。设计一套好的指令格式,不仅程序设计人员使用起来很方便,硬件实现起来比较容易,而且能够节省大量的程序存储空间。下面,举一个例子来说明。

表 2.15 列出了美国 Burroughs 公司生产的 B-1700 计算机系统,分别采用 8 位定长操作码、4-6-10 扩展操作码和全 Huffman 编码时,整个操作系统所用指令的操作码的总长度。从表中可以看出,改进操作码的编码方式可以节省大量的程序存储空间。

表 2.15 B-1700 计算机操作码编码方式比较

操作码编码方式	整个操作系统所用 指令的操作码总位数	改进的百分比
8 位定长编码	301 248	0
4-6-10 扩展编码	184 966	39%
Huffman 编码	172 346	43%

另外,改进地址码的设计方法,也可以节省大量的程序存储空间。

指令格式优化设计的主要目标有两个,一是节省程序的存储空间,二是指令格式要尽量规整,以减少硬件译码的复杂程度。另外,指令格式优化后,不应该降低指令的执行速度(例如,由于一条指令在主存储器中存放时,跨越了多个存储字,从而在读取指令时增加了访问存储器的次数)。

以下,主要介绍操作码和地址码的优化设计方法,最后,举例说明整个指令格式的优化设计方法。

2.3.1 指令的组成

指令一般由两部分组成,操作码和地址码。

操作码(OPC)	地址码(A)
----------	--------

操作码通常包括两部分内容:

1. 指令的操作种类,如运算操作(加、减、乘、除等)、数据传送、移位、转移、输入输出操作等。

2. 所用操作数的数据类型。如果采用自定义数据表示法,在操作码中不必指出操作数的数据类型,只需要指出指令的操作种类即可。

一个地址码通常包括三部分内容,而且,在一条指令中,经常包括有两个或两个以上

的地址码。

1. 操作数的地址。对于间接寻址方式,给出间接地址;对于立即数寻址方式,在地址码部分直接给出操作数;对于寄存器寻址方式,给出通用寄存器编号;对于变址寻址方式,给出变址寄存器编号。

2. 地址的附加信息。如偏移量、块长度、跳距等。

3. 寻址方式。如直接寻址、间接寻址、立即数寻址、变址寻址、自动增量寻址、自动减量寻址、相对寻址等。例如,在 PDP-11 计算机中,用 3 位表示 8 种寻址方式,在 VAX-11 计算机中,用 4 位表示 16 种寻址方式。

2.3.2 操作码的优化表示

操作码的表示方法通常有三种,固定长度操作码,Huffman 编码法和扩展编码法,下面分别介绍。

2.3.2.1 固定长操作码

一般处理机的指令条数通常为几十条至几百条,用一个字节(8 位)表示,非常规整,硬件译码也很简单。IBM 公司生产的许多大中型计算机,目前的许多 RISC(精简指令系统计算机)体系结构都采用这种编码方法。

固定长操作码的主要缺点是:浪费了许多信息量,即操作码的总长度增加了,从表 2.15 中可以看到,操作码的总长度要增加 40% 左右。

2.3.2.2 Huffman 编码法

Huffman 编码法是 1952 年由 Huffman 首先提出的一种编码方法,开始主要用于电报报文的编码。如 26 个英文字母中,e、t 等的使用频率最高,用短码表示;q、x 等的使用频率很低,用长码表示。这样,可以缩短整个报文的长度,减少报文的传送时间。Huffman 编码法也可以用在其它许多地方,如存储空间压缩和时间压缩等。

要采用 Huffman 编码法表示操作码,必须先知道各种指令在程序中出现的概率,这通常可以通过对已有典型程序进行统计得到。

根据 Huffman 编码法的原理,采用最优 Huffman 编码法表示的操作码的最短平均长度可以通过如下公式计算:

$$H = - \sum_{i=1}^n p_i \cdot \log_2 p_i$$

其中 p_i 表示第 i 种操作码在程序中出现的概率,一共有 n 种操作码。

最优 Huffman 编码法与固定长操作码编码方法相比,固定长操作码编码方法的信息冗余量可以表示为:

$$R = 1 - \frac{\sum_{i=1}^n p_i \cdot \log_2 p_i}{\lceil \log_2 n \rceil}$$

下面,我们通过一个例子来说明采用 Huffman 编码法来优化操作码的具体做法,并

计算操作码的平均长度和信息冗余量。

假设一台模型计算机共有 7 种不同的操作码。如果采用固定长度操作码表示,需要 3 位操作码。如果采用 Huffman 编码法,首先要知道各种操作码在程序中出现的概率,如表 2.16 所示。

采用最优 Huffman 编码法,操作码的最短平均长度为:

$$H = - \sum_{i=1}^7 p_i \cdot \log_2 p_i$$

把表 2.16 中的数据代入上面的公式,得到采用最优 Huffman 编码法时,操作码的最短平均长度:

$$\begin{aligned} H &= 0.45 \times 1.152 + 0.30 \times 1.737 + 0.15 \times 2.737 + 0.05 \times 4.322 \\ &\quad + 0.03 \times 5.059 + 0.01 \times 6.644 + 0.01 \times 6.644 \\ &= 1.95 \end{aligned}$$

表 2.16 模型机的操作码 Huffman 编码法

指令序号	出现的概率	Huffman 编码	操作码长度
I1	0.45	0	1 位
I2	0.30	1 0	2 位
I3	0.15	1 1 0	3 位
I4	0.05	1 1 1 0	4 位
I5	0.03	1 1 1 1 0	5 位
I6	0.01	1 1 1 1 1 0	6 位
I7	0.01	1 1 1 1 1 1	6 位

如果采用 3 位固定长操作码,与采用最优 Huffman 编码法表示操作码相比,信息的冗余量为 35%。

$$R = 1 - \frac{H}{\lceil \log_2 7 \rceil} = 1 - \frac{1.95}{3} \approx 35\%$$

实际上,操作码不可能达到最优 Huffman 编码法的最短平均长度,这是因为操作码的位数必须是正整数。然而,我们可以利用 Huffman 算法,通过构造 Huffman 树对操作码进行编码。上面这个例子的具体编码过程如图 2.19 所示。

利用 Huffman 树进行操作码编码的方法,又称为最小概率合并法。对于上面这个例子,具体的编码方法是:首先,把所有 7 条指令按照操作码在程序中出现的概率值,自左向右排列好,每条指令是一个结点;然后,选取两个概率最小的结点合并成一个概率值是二者之和的新结点,并把这个新结点插入到其它还没有合并的结点中间;再在新的结点集合中选取两个概率最小的结点进行合并,如此继续进行下去,直至全部结点都合并完毕,最后得到一个根结点,根结点的概率值为 1。

从图中可以看到,每个结点都有两个分支,分别用一位代码“0”和“1”表示。如果要得到某一条指令的操作码编码,可以从根结点开始,沿箭头所指方向,到达属于该指令的概

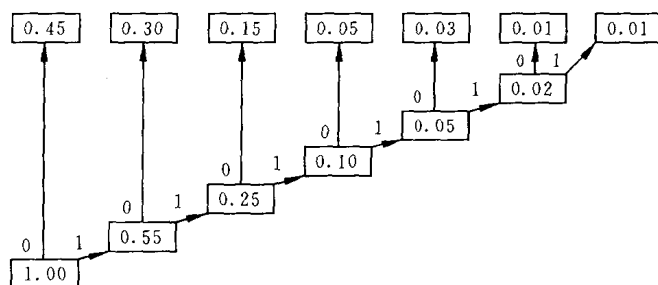


图 2.19 利用 Huffman 树进行操作码编码

率结点,把沿线所经过的代码组合起来得到这条指令的操作码编码。例如,要想得到概率值为 0.15 的指令的操作码编码,可以从概率值为 1.00 的根结点开始,沿箭头向右上方,到达概率值为 0.55 结点,于是得到第一位操作码“1”,继续向右上方,到达概率值为 0.25 结点,得到第二位操作码,也是“1”,最后沿向上的箭头,到达概率值为 0.15 的结点,得到最后一位操作码“0”,把所经过的代码组合起来得到概率值为 0.15 这条指令的操作码编码为“110”。

应当指出,采用上述方法形成的操作码编码并不是唯一的,只要将任意一个二叉结点上的“0”和“1”交换,就可以得到一组新的操作码编码。然而,无论怎样交换,操作码的平均长度是唯一的。

采用 Huffman 编码法所得到的操作码的平均长度为:

$$H = \sum_{i=1}^n p_i \cdot l_i$$

其中 p_i 表示第 i 种操作码在程序中出现的概率, l_i 表示第 i 种操作码的二进制位数,一共有 n 种操作码。

把具体数值代入,得到:

$$\begin{aligned} H &= \sum_{i=1}^7 p_i \cdot l_i = 0.45 \times 1 + 0.30 \times 2 + 0.15 \times 3 + 0.05 \times 4 \\ &\quad + 0.03 \times 5 + 0.01 \times 6 + 0.01 \times 6 \\ &= 1.97(\text{位}) \end{aligned}$$

与最优 Huffman 编码法相比,操作码的平均长度只长了 0.02 位,这种编码方法的信息冗余量仅为:

$$R = 1 - \frac{1.95}{1.97} \approx 1.0\%$$

与采用 3 位固定长操作码的信息冗余量 35% 相比要小得多。

2.3.2.3 扩展编码法

采用 Huffman 编码法能够使操作码的平均长度最短,信息的冗余量最小。然而,这种编码方法所形成的操作码很不规整。在上面这个例子中,7 条指令就有 6 种不同长度的操作码。这样,既不利于硬件的译码,也不利于软件的编译。另外,它也很难与地址码配合,

形成有规则长度的指令编码。

在许多处理机中,采用了一种新的折中的方法,称为扩展编码法。这种方法是由固定长操作码与 Huffman 编码法相结合形成的。

对于上一节中那个有 7 条指令的模型机的例子,如果采用扩展编码法编排操作码,可以有多种方法。举两个例子,如果要求操作码最长不得超过 5 位,则可以采用 1-2-3-5 扩展编码法。如果要求操作码最长不得超过 4 位,则可以采用 2-4 等长扩展编码法。编码结果如表 2.17 所示。

表 2.17 模型机的操作码扩展编码法

指令序号	出现的概率	1-2-3-5 扩展编码	2-4 等长扩展编码
I1	0.45	0	0 0
I2	0.30	1 0	0 1
I3	0.15	1 1 0	1 0
I4	0.05	1 1 1 0 0	1 1 0 0
I5	0.03	1 1 1 0 1	1 1 0 1
I6	0.01	1 1 1 1 0	1 1 1 0
I7	0.01	1 1 1 1 1	1 1 1 1

采用 1-2-3-5 扩展编码法的操作码最短平均长度为:

$$H = 0.45 \times 1 + 0.30 \times 2 + 0.15 \times 3 + (0.05 + 0.03 + 0.01 + 0.01) \times 5 \\ = 2.00$$

信息冗余量为:

$$R = 1 - \frac{1.95}{2.00} = 2.5\%$$

采用 2-4 等长扩展编码法的操作码最短平均长度为:

$$H = (0.45 + 0.30 + 0.15) \times 2 + (0.05 + 0.03 + 0.01 + 0.01) \times 4 \\ = 2.20$$

信息冗余量为:

$$R = 1 - \frac{1.95}{2.20} = 11.4\%$$

为了便于实现分级译码,一般采用等长扩展法,例如,操作码按 4 位、8 位、12 位,每次加长 4 位的扩展方法,记作 4-8-12 扩展法,还有 3-6-9 扩展法等。当然,也可以根据具体需要,采用每次扩展的位数不等的不等长扩展法。

对于等长扩展法,根据采用不同的扩展标志还可以有多种不同的扩展方法。例如,对于 4-8-12 扩展法,有采用保留一个码点标志的 15/15/15……扩展法,采用每次保留一个标志位的 8/64/512……扩展法等。当然,也可以根据不同的需要,在操作码长度扩展过程中,每次采用不同的扩展标志。

对于 4-8-12 等长扩展法中的 15/15/15……扩展法和 8/64/512……扩展法,操作码

的具体编码方法如图 2.20 所示。

操作码编码	说 明	操作码编码	说 明
0000	4 位长度的 操作码共 15 种	0000	4 位长度的 操作码共 8 种
0001		0001	
...		...	
1110		0111	
1111 0000	8 位长度的 操作码共 15 种	1000 0000	8 位长度的 操作码共 64 种
1111 0001		1000 0001	
...		...	
1111 1110		1111 0111	
1111 1111 0000	12 位长度的 操作码共 15 种	1000 1000 0000	12 位长度的 操作码共 512 种
1111 1111 0001		1000 1000 0001	
...		...	
1111 1111 1110		1111 1111 0111	

(a) 等长 15/15/15……扩展法

(b) 等长 8/64/512……扩展法

图 2.20 操作码的等长扩展编码法

在设计操作码时,具体采用哪种扩展编码方法,要根据所设计系统中各种指令出现的概论分布情况来决定。例如,某系统中,有 15 种指令出现的概论值比较大,另 15 种次之,其余指令出现的概论值很小,这时可采用 15/15/15……扩展编码法。如果有 8 种指令出现的概论值比较大,另 64 种指令出现的概论值次之,可采用 8/64/512……扩展编码法。总之,无论采用哪一种扩展编码方法,衡量的标准是要看这种编码方法的操作码平均长度是否最短,或信息的冗余量是否最小。

另外,也可以根据所设计计算机系统的具体情况,采用不等长扩展编码方法,例如,采用 4-6-10 扩展编码法,根据所设计系统中各种指令出现的概论分布情况,可以有很多种编码方法,如表 2.18 所示。

表 2.18 4-6-10 不等长扩展编码法举例

编码方法	各种不同长度操作码的指令种类			总的指令种类
	4 位操作码	6 位操作码	10 位操作码	
15/3/16	15	3	16	34
8/31/16	8	31	16	55
8/30/32	8	30	32	70
8/16/256	8	16	256	280
4/32/256	4	32	256	292

2.3.3 地址码的优化表示

地址码在指令中所占的长度最长,其编码长度主要与地址码的个数、操作数所存放的存储设备(通用寄存器、主存储器、堆栈等)、存储设备的寻址空间大小、编址方式、寻址方式等有关。其中,有些问题已经在前面介绍过了,这里主要介绍地址个数的选择和单个地

址码的长度如何优化两个问题。

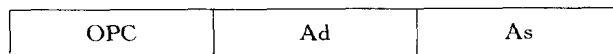
2.3.3.1 地址码个数的选择

在目前的计算机系统中,地址码的个数通常有三个、两个、一个及没有地址码等四种情况,其指令格式如下。

1. 三地址指令。有两个源操作数地址,一个目标地址。这种指令在运算后,不会破坏任何一个操作数。



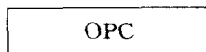
2. 二地址指令。有一个源操作数地址,一个目标地址。运算所需的两个操作数分别来自源地址和目标地址,运算结果覆盖掉目标地址中的源操作数。



3. 一地址指令。只有一个源地址。运算所需的两个操作数,一个隐含来自累加器(通用寄存器),另一个来自源地址,运算结果隐含放入累加器。采用一地址指令的计算机系统必须有,而且只能有一个累加器。



4. 零地址指令,即堆栈型计算机指令。运算所需操作数隐含从堆栈顶部弹出,运算结果隐含压入堆栈顶部,因此不需要地址码。



下面,通过一个典型算术表达式的计算,来评价以上四种指令(不同地址码个数)的优缺点及所适用的场合。即在设计计算机指令系统时,如何选择地址码的个数。

评价的标准主要有两个:一是程序的存储量,即程序中所有指令的长度的总和最短;二是程序的执行速度,即程序在执行过程中访问主存储器的信息(包括指令和数据)量的总和最短。

例 2.9 计算算术表达式: $x = \frac{a * b + c - d}{e + f}$ 。

在下面的程序中,用大写字母 A 表示数据 a 的地址,其它依次类推,用 R1、R2 等表示通用寄存器(或称为累加器),用 Y 等表示存放中间结果的存储单元。所有程序都用直接寻址方式编写。

用三地址指令编写的程序如下:

```
MUL    X, A, B    ;X 单元暂时用来存放中间运算结果
ADD    X, X, C
```

```

SUB    X, X, D    ;X 单元中存放的是分子运算的结果
ADD    Y, E, F    ;计算分母
DIV    X, X, Y    ;最后运算结果在 X 单元中

```

用二地址指令编写的程序如下：

```

MOVE   X, A      ;复制一个临时变量到 X 单元中
MUL    X, B
ADD    X, C
SUB    X, D      ;X 单元中存放的是分子运算的结果
MOVE   Y, E      ;复制一个临时变量到 Y 单元中
ADD    Y, F      ;Y 单元中存放的是分母运算的结果
DIV    X, Y      ;最后运算结果在 X 单元中

```

用多累加器(通用寄存器)的二地址指令编写的程序如下：

```

MOVE   R1, A     ;把操作数 a 取到 R1 通用寄存器中
MUL    R1, B
ADD    R1, C
SUB    R1, D     ;通用寄存器 R1 中存放分子运算结果
MOVE   R2, E
ADD    R2, F     ;通用寄存器 R2 中存放分母运算结果
DIV    R1, R2    ;最后运算结果在通用寄存器 R1 中
MOVE   X, R1     ;把最后运算结果存入 X 单元中

```

用一地址指令编写的程序如下：

```

LOAD   E        ;最好先做分母,取一个操作数到累加器中
ADD    F        ;分母运算结果在累加器中
STORE  X        ;保存分母运算结果,腾出累加器
LOAD   A        ;取分子的一个操作数到累加器中
MUL    B
ADD    C
SUB    D        ;累加器中是分子运算结果
DIV    X        ;最后运算结果在累加器中
STORE  X        ;保存最后运算结果到 X 单元中

```

用零地址指令编写程序时,首先要把这个算术表达式转换成逆波兰表达式,结果为：

$ab * c + d - ef + /$,程序如下：

```

PUSH   A        ;把操作数 a 压入堆栈
PUSH   B        ;操作数 b 压入堆栈
MUL                    ;弹出堆栈顶部的两个操作数做乘法,结果压入堆栈
PUSH   C
ADD
PUSH   D

```

SUB ;栈顶是分子运算的结果
 PUSH E
 PUSH F
 ADD
 DIV ;栈顶是最后运算的结果
 POP X ;保存最后运算结果

为了便于比较,把以上 5 个用不同地址数指令编写的程序的有关参数列表,如表 2.19 所示。其中,指令条数和访存次数只是相对参数,这是因为,对于各种不同地址数的指令,其本身的长度可能相差很大,例如,一条三地址指令的长度可能是一条零地址指令长度的 10 倍左右。

表 2.19 各种不同地址数指令的程序存储量和执行速度比较

地址数目	指令条数	访存次数	程序存储量	执行速度(访存信息量)
三地址	5	20	$5P+15A=65B$	$5P+15A+15D=185B$
二地址	7	26	$7P+14A=63B$	$7P+14A+19D=215B$
一地址	9	18	$9P+9A=45B$	$9P+9A+9D=117B$
零地址	12	41	$12P+7A=40B$	$12P+7A+29D=272B$
二地址 R 型	8	15	$8P+7A+9R=40B$	$8P+7A+9R+7D=96B$

实际上,人们最关心的是程序的存储量和程序的执行速度。为了便于比较,在表 2.19 中,程序的存储量都统一转换成用字节数来表示。对于程序的执行速度,在一般计算机系统中,与访问存储器的次数的关系最大,包括取指令,读源操作数及写运算结果,因此,用程序访问存储器的信息量来表示是合理的。为了便于比较,在表中也把它统一转换成用字节数来表示。

在表 2.19 中,“P”表示操作码长度,“5P”则表示 5 个操作码的长度,同样,“A”表示地址码长度,“D”表示数据长度,“R”表示通用寄存器的地址码长度,最后的“B”表示字节数,“65B”表示这个程序的总存储量是 65 个字节。另外,“二地址 R 型”是指二地址的通用寄存器型指令。

在一般计算机系统中,操作码的长度为 5 至 8 位,地址码的长度为 16 至 32 位,一个数据的长度为 32 至 64 位,通用寄存器地址的长度为 3 至 6 位。为了便于比较,在表 2.19 中取关系: $D=2A=8P=16R=8B$,即一个数据长度为 8 个字节,一个地址码的长度(包括地址或地址的偏移量,变址寄存器,基址寄存器,寻址方式及其它寻址信息)为 4 个字节,一个操作码的长度为 1 个字节,一个通用寄存器地址的长度为半个字节。转换结果,用字节数表示的 5 个程序的程序总存储量和程序执行速度(访问主存储器的信息量),填在表 2.19 中等号之后。

从表 2.19 中可以看出,用各种不同地址数指令编写的程序,它们的特点及适用场合,归纳比较如表 2.20 所示。

表 2.20 各种不同地址数指令的特点及适用场合

地址数目	指令长度	程序存储量	程序执行速度	适用场合
三地址	短	最大	一般	向量,矩阵运算为主
二地址	一般	很大	很低	一般不宜采用
一地址	较长	较大	较快	连续运算,硬件结构简单
零地址	最长	最小	最低	嵌套,递归,变量较多
二地址 R 型	一般	最小	最快	多累加器,数据传送较多

通过以上分析,可以得出如下结论:

1. 对于二地址结构,只有采用多累加器(通用寄存器)才是可取的。如果没有通用寄存器的支持,对于一般的二地址结构,无论是程序的存储量,还是程序的执行速度都很差,一般不宜采用。

另外,二地址指令天然的优点是用于只需要一个源操作数的场合,如数据传送操作等。对于需要两个操作数的场合,运算结果要覆盖掉其中的一个操作数,因此,对于向量、矩阵等运算的支持不好。

2. 如果要求硬件结构简单,并且以连续运算为主,如求累加和,算术表达式计算等,宜采用一地址结构。

3. 对于以向量、矩阵运算为主的计算机系统,最好采用三地址结构。对于三地址结构的计算机,可以采用通用寄存器。但是,由于向量寄存器的硬件结构比较复杂,因此,有些向量计算机直接在主存储器中运算。

4. 对于解决以递归问题为主的计算机系统,宜采用零地址结构。由于零地址指令访问堆栈的信息量很大,从而造成程序执行速度降低。为了克服这一缺点,可以把堆栈顶部改成一个高速寄存器堆。

2.3.3.2 缩短地址码长度的方法

目前,计算机系统的主存储器容量通常都很大,而且越来越大。另外,由于普遍采用了虚拟存储器结构,要求指令中给出的地址码是一个虚拟地址(逻辑地址),其长度比实际主存储器的容量所要求的长度还要长得多。例如,目前一般计算机系统虚拟地址空间为 4GB 左右,因此,要求一个地址码的实际长度为 32 位左右。对于多地址结构的指令系统而言,如此长的地址码是无法容忍的。因此,如何缩短地址码的长度,是设计指令系统时必须予以考虑的一个问题。

由于在一般计算机系统中,逻辑地址空间的大小是确定的,因此,缩短地址码长度的根本目的是要用一个比较短的地址码表示一个比较大的逻辑地址空间,同时也要求有比较灵活有效的寻址方式。

缩短地址码长度的方法很多,现举几个常用的例子。

1. 用间址寻址方式缩短地址码长度。在主存储器的低端开辟出一个专门用来存放地址的区域,由于表示存储器低端部分的地址所需要的地址码长度可以很短,而一个存储字

(一次访问存储器所能获得的数据)的长度通常与一个逻辑地址码的长度相当。如果一个存储字的长度短于一个逻辑地址的长度,也可以用几个连续的存储字来存放一个逻辑地址码。例如,在主存储器最低端的 1KB 之内有一个用来存放地址码的区域,如果主存储器是按字节编址的,并且,一个存储字的长度为 32 位,那么,在指令中只要用 8 位(256 个存储字即为 1K 字节)长度就能表示一个 32 位长的逻辑地址,即使再加上寻址方式等信息,一个地址码的长度也只有十多位。

2. 用变址寻址方式缩短地址码长度。由于程序的局部性,在变址寻址方式中使用的地址偏移量可以比较短,例如在 IBM370 系列机中为 12 位。通常可以把比较长的基地址(如 32 位)放在变址寄存器中,在指令的地址码中只需给出比较短的地址偏移量。因此,采用变址寻址方式的地址码长度通常有十多位或二十位左右就可以了。

3. 用寄存器间接地址方式缩短地址码长度是最有效的方法。由于寄存器的数量比较少,通常表示一个寄存器的地址只需要很少几位,而一个寄存器的字长足可以放下一个逻辑地址。例如,有 8 个用于间接寻址的寄存器,每个寄存器的长度是 32 位,这样,用一个 3 位的地址码就能表示一个 32 位的逻辑地址,再加上寻址方式等信息,一个地址码的长度也不超过 10 位。

用来支持间接寻址的寄存器,可以借用通用寄存器,也可专门设置。

2.3.4 指令格式设计举例

指令的长度有固定长度的和可变长度的两种,有些 RISC(精简指令系统计算机)的指令是固定长度的,但是,由于各种不同用途的指令所要表达的信息量差别很大,因此,目前多数计算机系统的指令是可变长度的。

在目前的绝大多数计算机系统中,指令长度通常取 8 的倍数,并且要求,当指令长度小于一个存储字时,不要跨越存储字存放。这是因为,如果跨越两个存储字存放,读取一条指令就需要访问两次存储器,从而降低了指令的执行速度。

操作码长度也有固定长度的和可变长度的两种,下面分别举两个例子。

IBM370 系列计算机的指令长度有 16 位、32 位和 48 位等几种,所有指令的操作码都是 8 位固定长度。在操作码中要给出操作种类、数据类型和寻址方式等。主要指令的格式如图 2.21 所示。

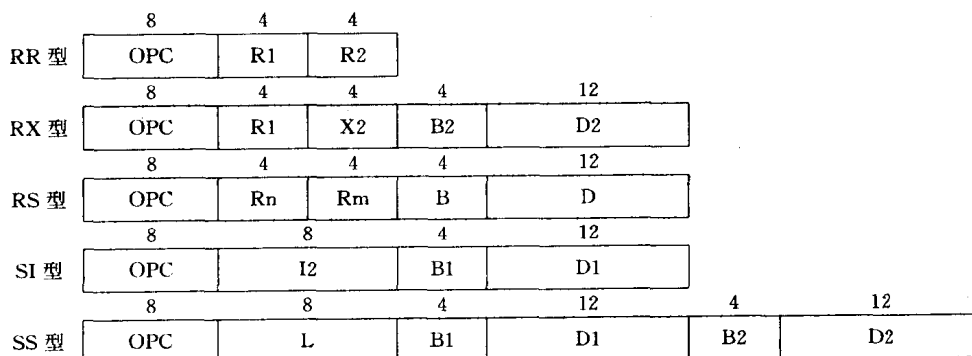


图 2.21 IBM 370 系列计算机的指令格式

地址个数采用两地址。有 16 个通用寄存器,可兼做变址寄存器和基址寄存器使用。如果参加运算的操作数都在通用寄存器中,运算结果也放在通用寄存器中,则为 RR 型指令,指令字长只需要 16 位。一个存储字中可以放两条指令。如果参加运算的一个操作数来自主存储器,并采用变址寻址方式,另一个操作数来自通用寄存器,运算结果也放在通用寄存器中,则为 RX 型指令,指令字长 32 位,正好一个存储字。

另外,RS 型指令可用来在通用寄存器与主存储器之间一次传送多个数据。 R_n 为起始通用寄存器编号, R_m 为结束通用寄存器编号,只要用一条指令就能把通用寄存器中从 R_n 开始到 R_m 结束的所有数据都存入相邻的主存单元中,同样,也可以从主存储器的相邻单元读出多个数据到通用寄存器中。在程序调用和中断处理时,用这种指令可以快速保存和恢复程序现场。

SI 型指令的一个操作数在主存储器中,另一个操作数为立即数。立即数的长度为 8 位,当然,立即数只能是第二操作数。SS 型指令的两个操作数都在主存储器中,并且支持“串”操作。如字符串运算,十进制运算等,也可以用来把长度不超过 256 个字节的数据块从主存储器的一个区域搬到另一个区域。

小型计算机 PDP-11 的指令格式如图 2.22 所示。其指令长度和操作码长度都是可变的,所采用的寻址方式很有特点。

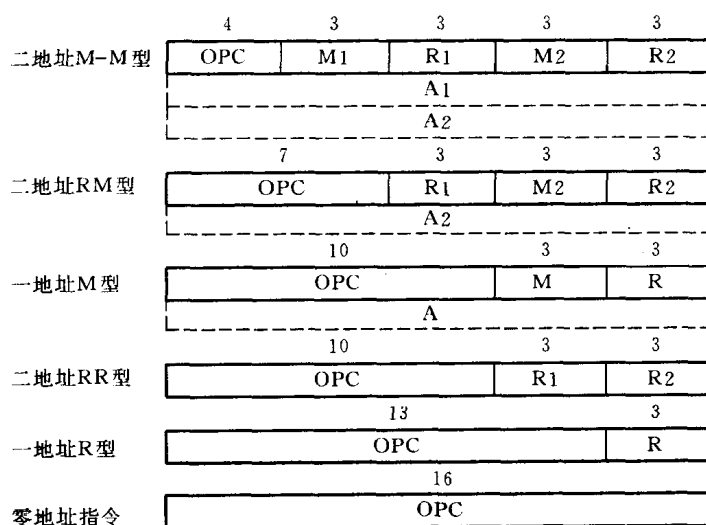


图 2.22 PDP-11 小型计算机的指令格式

操作码的长度有 4 位、7 位、10 位、13 位和 16 位等几种,从 4 位开始,按等长 3 位扩展。共有 8 个通用寄存器,其中也包括程序计数器(R_7)和堆栈指针(R_6)。寻址方式有 8 种,见表 2.21 所示。

从表 2.21 中可以看出,每个访问主存储器的地址只需要用 6 位表示,其中 3 位是寄存器编号,3 位是寻址方式。用 6 位就能寻址整个主存储器的逻辑地址空间。

表 2.21 PDP-11 计算机的寻址方式

代码	名 称	汇编符号	有效地址	R 中内容	说 明
000	寄存器寻址	R	R	不变	R 中内容即为操作数
001	自增寻址	(R)+	(R)	(R)+d	字指令 d=2, 字节指令 d=1
010	自减寻址	-(R)	(R)-d	(R)-d	d 的取值同上, (R) 先减后用
011	变址寻址	A(R)	(R)+A	不变	A 为偏移量, 16 位
100	寄存器间址	(R)	(R)	不变	
101	自增间址	@(R)+	((R))	(R)+d	d 的取值同上
110	自减间址	@-(R)	((R)-d)	(R)-d	d 的取值同上
111	变址间址	@A(R)	((R)+A)	不变	d 的取值同上, A 为偏移量

2.4 指令系统的功能设计

如果把计算机系统所要实现的任务分解成一个个基本功能,那么,在这些基本功能中,实际上只有极少数几种基本功能是必须用硬件的指令系统来实现的,而绝大多数基本功能既可以用硬件的指令系统来实现,也可以用软件的一段子程序来实现。

体系结构设计者在决定哪些基本功能用指令来实现时,主要考虑的因素有三个:速度、价格和灵活性。

用硬件的指令来实现,速度高、价格贵、灵活性差。

用软件的子程序来实现,速度低、价格便宜、灵活性好。

设计指令系统时,在功能方面的最基本要求是:指令系统的完整性、规整性、高效率 and 兼容性。

完整性是指作为通用计算机所应该具备的基本指令种类,在下一节中将介绍通用计算机的 5 类基本指令。

指令系统的规整性设计是硬件设计(如 VLSI 技术)和软件设计(如编译程序等)的需要。尽可能减少、甚至不出现任何例外情况和特殊用法,让所有运算部件都能对称、均匀地在所有数据存储单元(如主存储器、通用寄存器和堆栈)之间进行操作,对所有数据存储单元都能同等对待,无论是操作数或是运算结果都可以无任何约束地存放在任意数据存储单元中。

规整性主要包括对称性和均匀性。对称性是指各种与指令系统有关的数据存储设备的使用、操作码的设置等都要对称。例如,所有通用寄存器要同等对待。这一点在目前的许多计算机系统中都没有做到,往往隐含规定某一个或某几个通用寄存器有特殊的用途。在操作码的设置上,如果设置有 $A-B$ 指令,也应该设置 $B-A$ 指令;同样,如果有 $A \div B$ 的指令,也应该有 $B \div A$ 的指令。对于两个地址码的指令,由于运算结果要破坏掉其中的一个源操作数,即使是加法、乘法等指令最好也设置两种,如 $A+B \rightarrow A$ 和 $A+B \rightarrow B$ 。均匀

性是指对于各种不同的数据类型、字长、操作种类和数据存储设备(通用寄存器、主存储器、堆栈及输入输出设备等),指令的设置要同等对待。例如,某机器有5种数据表示(定点数、逻辑数、浮点数、十进制数、字符串),4种字长(单字长、双字长、半字长、字节),8种数据存储设备的有效排列(通用寄存器、主存储器、堆栈及它们之间有用的排列),则设计加法指令时,指令种类应该有: $5 \times 4 \times 8 = 160$ 种两地址加法指令。如果再考虑对称性的要求和不同寻址方式的要求等,指令种类还要增加很多倍,这实际上很难做到,也是不现实的。

因此,在设计指令系统时,对于规整性的要求必须有所选择。在RISC(精简指令系统计算机)体系结构中,规定运算型指令都在通用寄存器内进行操作,即使在CISC(复杂指令系统计算机)体系结构中,如果采用二地址通用寄存器结构,通常也规定,两个地址中必须有一个是通用寄存器。另外,对于逻辑数、十进制数、字符串等数据表示,双字长、半字长和字节等数据长度,可以适当减少指令种类。这样,可以把加法指令的种类压缩在10种之内。另外,如果采用自定义数据表示方式,每个数据只要带有几个标志位,规整性问题就不难解决了。

高效率是指指令的执行速度要快,使用频度要高。在RISC体系结构中,大多数指令都能在一个节拍内做完,而且只设置那些使用频度高的指令。而在CISC体系结构中,有些指令的执行速度非常低,需要几十个节拍才能完成,甚至是不固定节拍的(由所使用的数据决定指令执行所需要的节拍数,如检索、匹配、数据块搬家等指令)。体系结构设计者在设置这些低速度指令时要特别慎重。要考虑是用软件的一段子程序来实现,还是用硬件的一条指令来实现。对于那些使用频度比较低的指令,也要尽量少设置。在考虑指令的实现方法时,对于那些比较复杂但又必须设置的指令,可以采用微程序来实现,以减少硬件的复杂程度。总之,指令系统的设计要做大量的统计和模拟的工作,要在反复比较的基础上才能最后确定下来。

兼容性是计算机系统的生命力之所在。没有兼容性,大量的系统软件和更大量的各种应用软件就无法继承,计算机也就没有了市场。因此,在设计指令系统时,兼容性是必须考虑的,下面有一节还要具体介绍实现兼容性的几种方法。

2.4.1 基本指令系统

在设计通用计算机时,指令系统的完整性是必须要考虑的。数据传送类指令、运算类指令、程序控制类指令、输入输出指令、处理机控制和调试指令,这5类指令属于通用计算机系统的基本指令。

1. 数据传送类指令

数据传送指令的种类由如下三个主要因素决定:

(1) 数据存储设备的种类。数据传送指令的目标是要在相同或不同的数据存储设备之间传送数据。计算机系统中可以寻址的主要数据存储设备有通用寄存器、主存储器、堆栈等三种,这三种存储设备的有效排列有8种可能。对于输入输出设备,虽然它也可能是数据存储设备,但是,输入输出设备具有许多与其它数据存储设备不同的地方,因此,许多计算机系统用专门的输入输出指令对它进行管理。

(2) 数据传送的单位,通常有字、字节、数据块等 3 种。

(3) 采用的寻址方式,根据机器种类不同,差别很大,一般在 4 至 20 种之间。寻址方式可以放在操作码中,如 IBM370 计算机,也可以放在地址码中。如果把寻址方式放在地址码中,指令的种类就可以减少许多。

如果把寻址方式放在地址码中,并且不考虑输入输出设备,则以字为传送单位的指令应该有如下 8 种:

通用寄存器→通用寄存器

通用寄存器→主存储器

通用寄存器→堆栈

主存储器→通用寄存器

主存储器→主存储器

主存储器→堆栈

堆栈→通用寄存器

堆栈→主存储器

对于以字节或以数据块为单位的数据传送指令,指令种类可以适当减少。

2. 运算类指令

目前,在许多应用领域中,计算机的主要任务还是做运算(包括数据计算和符号处理),因此,运算型指令在整个指令系统中应该占有比较大的比重,如指令种类不少于 30%。如果所占比重过小,就会影响整个计算机系统的性能。

设计运算类指令时主要考虑如下四个因数的组合:

(1) 操作种类:包括加、减、乘、除等常用操作,另外还有比较、移位、检索、转换、匹配、清除、设置等操作;

(2) 数据表示:有定点数、浮点数、逻辑数、十进制数、字符串、定点向量、浮点向量等;

(3) 数据长度:字、双字、半字、字节、位等;

(4) 数据存储设备:主要有通用寄存器、主存储器和堆栈等三种。

例如,仅加法指令,且源操作数和运算结果都在通用寄存器中,一般应设置如下几种不同的指令:

寄存器-寄存器型的定点单字长加法指令,

寄存器-寄存器型的定点双字长加法指令,

寄存器-寄存器型的定点半字加法指令,

寄存器-寄存器型的字节加法指令,

寄存器-寄存器型的浮点单字长加法指令,

寄存器-寄存器型的浮点双字长加法指令,

寄存器-寄存器型的单字长逻辑加法(按位加法)指令,

寄存器-寄存器型的定点向量加法指令,

寄存器-寄存器型的浮点向量加法指令。

对于两地址指令,如果再要考虑数据的存储设备,除了 RR 型指令外,还有 RS 型、SS

型、堆栈型等组合,仅加法指令就有几十种之多。

因此,在对上述这些因素进行组合时,必须考虑指令的执行时间、使用频度、硬件实现的复杂程度等多方面的情况。分析出哪些指令是必须要设置的,哪些是可有可无的(也可用软件的一段子程序来实现),哪些是不合理或不应该设置的。并经过模拟实验和统计分析,最后确定出合理的运算类指令。

另外,有两类比较特殊的操作,移位指令和位/串操作指令的设置与一般的运算类指令有所不同。

对于移位指令,要组合以下三个因素:

- (1) 移位方向,有左移(L)和右移(R)两种;
- (2) 移位种类,有算术移位(A)、逻辑移位(L)和循环移位(R)三种;
- (3) 移位长度,有单字长(S)和双字长(D)两种。

把以上三个因素进行组合: $3 \times 2 \times 2 = 12$ 种。因为逻辑左移和算术左移的操作结果完全相同的,从而可以减少两种,所以,在一般计算机系统中,移位指令应该有10种。它们分别是:

SLAS	单字长算术左移
SRAS	单字长算术右移
SLLS(SLAS)	单字长逻辑左移,或单字长算术左移
SLRS	单字长循环左移
SRRS	单字长循环右移
SLAD	双字长算术左移
SRAD	双字长算术右移
SLLD(SLAD)	双字长逻辑左移,或双字长算术左移
SLRD	双字长循环左移
SRRD	双字长循环右移

移位指令一般应该有两个源操作数,其中,第二源操作数给出移位的位数。

位操作通常有:置位(把一个字或一个字符串中某一位置“1”)、清位、位测试、找位(把一个字或一个字符串中第一个“1”的序号找出来)。

字符串操作主要有:比较、查找、匹配、转换(如ASCII码与BCD码的转换,ASCII码与EBCDIC码的转换等)。

3. 程序控制指令

程序控制指令主要包括三类,转移指令(包括无条件转移和有条件转移)、程序调用和返回指令、循环控制指令。其中,前两类指令在一般计算机中是必须具备的。最后一类指令用于对循环程序进行优化。

转移指令中的无条件转移指令通常有两种:一种时是局部无条件转移,采用相对寻址方式,转移范围一般在+127到-128之间;另一种是全局无条件转移,可以在整个寻址空间内转移。

条件转移指令所依据的转移条件主要有:全零(Z)、正负号(N)、进位(C)、溢出(V)及它们的组合等。主要条件转移指令有:

BEQ	等于零转移
BNEQ	不等于零转移
BLS	小于转移
BGT	大于转移
BLEQ	小于等于转移,或不大于转移
BGEQ	大于等于转移,或不小于转移
BLSU	不带符号小于转移
BGTU	不带符号大于转移
BLEQU	不带符号小于等于转移,或不带符号不大于转移
BGEQU	不带符号大于等于转移,或不带符号不小于转移
BCC	没有进位转移
BCS	有进位转移
BVC	没有溢出转移
BVS	有溢出转移

第二类程序控制操作是程序调用和返回指令,主要有两条:

CALL	转入子程序
RETURN	从子程序返回

这两条指令本身可以带有条件,当测试条件满足时转入子程序或从子程序返回,也可以不带条件。如果调用和返回指令本身不带条件,则要与条件转移指令联合使用,如果本身带有条件,目标程序就可以简化。

在执行调用指令时,要保存硬件现场(主要指程序计数器和处理机状态字)和软件程序现场(指在子程序中要使用的通用寄存器等);当从子程序返回时,再恢复这些现场。在设置有系统堆栈的计算机中,硬件现场和程序现场都可以压入堆栈。如果没有设置堆栈,则要在主存储器中开辟出一块专门的区域或指定专门的通用寄存器等来保存硬件现场。而软件程序现场的保存则由程序员自行决定。

另外,中断控制指令和自陷指令(或称为过程调用指令)也属于程序调用指令。中断控制指令主要有:开中断、关中断、改变屏蔽状态、从中断程序返回等指令。自陷指令主要用来转入例行子程序,或在程序调试过程中用来设置断点。

4. 输入输出指令

输入输出指令通常比较简单,采用单的直接寻址方式,数据字长一般以字节为单位。主要的输入输出操作有:启动设备、停止设备、测试设备、对设备进行控制及数据的输入或输出操作等。

在多用户或多任务环境下,输入输出指令属于特权指令。当程序需要进行输入输出操作时,用系统调用进入操作系统,由操作系统对设备统一进行管理。

有些计算机系统把输入输出设备与主存储器统一编址,共用同一个零地址空间。在这类计算机系统中,没有专门的输入输出指令,所有能够访问主存储器的指令都能访问输入输出设备。

5. 处理机控制和调试指令

在一般的计算机系统中,处理机有两个状态:管态和用户态,或称主态和从态。这两个

状态需要互相切换,而且,这两个状态下所能使用的指令应该有所区别。例如,一般用户应该严禁使用处理机状态切换、系统资源分配和管理等指令,否则,一个多用户操作系统将无法正常工作。

在一般通用计算机系统中,按照指令的使用权限,可以把指令分为两大类:一般指令和特权指令。只有系统管理程序能够使用,一般用户程序不能使用的指令称为特权指令。特权指令主要包括处理机状态的设置和管理、系统硬件和软件资源的管理、进程的管理等。

只有在管态下才能够使用机器所提供的全部指令,包括特权指令。在用户态下,只能使用一般指令,不能使用特权指令。有些处理机还设置有更多的状态,如 VAX-11 处理机有四种状态,每种状态有不同的特权,离内核态越近,所具有的特权就越高。这四种状态分别是:

内核态(K):供操作系统的核心使用,运行操作系统内核程序,包括存储器页面管理和调度,输入输出子系统及大部分系统服务;

执行态(E):运行部分系统服务程序,包括系统调用,系统的记录和管理等;

管理态(S):用于命令解释等服务,运行命令解释程序;

用户态(U):用于一般用户服务,运行用户级程序,包括各种实用程序、编译程序和调试程序等。

调试指令主要用于硬件和软件的调试。硬件调试指令主要有:钥匙位置的读取,开关状态的读取,内部主要寄存器和主存单元的显示等。软件调试指令主要有断点的跟踪和自陷阱指令等。

一般处理机中都设置有调试指令,但这些指令对一般用户是不公开的。

2.4.2 复杂指令系统(CISC)

目前,指令系统的优化设计有两个截然相反的方向。一个是增强指令的功能,设置一些功能复杂的指令,把一些原来由软件实现的、常用的功能改用硬件的指令系统来实现,这种计算机系统称为复杂指令系统计算机(complex instruction set computer, CISC)。另一个是八十年代新发展起来的,尽量简化指令功能,只保留那些功能简单,能在一个节拍内执行完成指令,较复杂的功能用一段子程序来实现,这种计算机系统称为精简指令系统计算机(reduced instruction set computer, RISC)。本节介绍复杂指令系统,精简指令系统在下一节中介绍。

2.4.2.1 目标程序的优化

目标程序是由机器指令直接组成的,是要在处理机直接执行的,因此,面向目标程序优化指令系统是提高计算机系统性能的最直接的办法。优化目标程序的指标主要有两个:一是缩短程序的长度,即减少程序的空间开销;另一个是缩短程序的执行时间,即减少程序的时间开销。

优化目标程序的方法是:对大量的目标程序及其执行情况进行统计分析,找出那些使用频度高、执行时间长的指令或指令串。对于那些使用频度高的指令,用硬件加快其执行,

就能缩短整个程序的执行时间。对于那些使用频度高的指令串,用一条新的指令来代替它,这样,不但能缩短整个程序的执行时间,而且能缩短整个程序的长度,从而减少程序的空间开销。

优化目标程序的主要途径有以下几个方面。

1. 增强数据传送指令的功能

我们用 C 编译程序和 PROLOG 解释程序对 Intel 8088 处理机的所有指令的使用频度和执行时间进行了统计,其中,最常用的 3 种数据传送指令的使用频度和执行时间的情况统计如表 2.22 所示。

从表 2.22 中看出,Intel 8088 处理机的 3 种数据传送指令的使用频度在程序中占近 40%,执行时间占 30%以上。从 IBM 大中型计算机的统计结果看,数据传送指令所占的比例还要高。因此,数据传送指令在整个指令系统中占有非常重要的地位,设计好数据传送指令对提高计算机系统的性能至关重要。

表 2.22 Intel 8088 处理机主要数据传送指令的使用情况

指令种类	使用频度 %	执行时间 %
MOV	24.85	17.44
PUSH	10.36	11.11
POP	4.14	2.61
合计	39.35	31.16

在通用寄存器与通用寄存器之间,通用寄存器与主存储器之间,主存储器与主存储器之间设置数据块传送指令,是对向量和矩阵运算的有力支持。如在 IBM370 计算机中有这样一条指令:

8	4	4	4	12
OPC	R1	R3	B2	D2

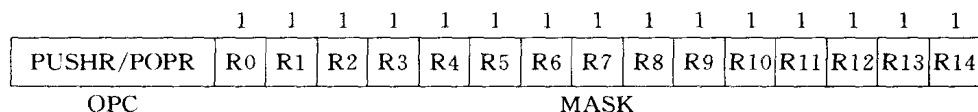
对于通用寄存器,数据块的起始地址是 R1,结束地址是 R3,对于主存储器,数据块的起始地址是 (B2)+D2。有这样的两种指令,可以分别把数据块从通用寄存器传送到主存储器,或者相反。

在 IBM370 计算机中还有一种指令,用一条指令就能在主存储器与主存储器之间传送一个字节数不大于 256 的数据块。这种指令对于向量、矩阵、字符串和十进制运算都有很好的支持。

8	8	4	12	4	12
OPC	L	B1	D1	B2	D2

第二操作数地址是源数据块地址,其起始地址为 (B2)+D2,数据块长度为 L,第一操作数地址是目标数据块地址,其起始地址为 (B1)+D1

在通用寄存器与堆栈之间设置数据块传送指令,能够在程序调用和程序中断时,快速保存和恢复程序现场。如 VAX-11 计算机中有这样两种指令。



PUSHR 指令能够把所有屏蔽字为 1 的位所对应的通用寄存器内容压入堆栈,POPR 指令做相反的操作。R15 即为程序计数器,不参与这种指令的操作。

2. 增强运算型指令的功能

在科学计算的应用程序中,经常要计算各种各样的函数,为此,在有些计算机系统设置有常用的函数运算指令。例如,开平方 \sqrt{x} ,三角函数 $\sin(x)$ 、 $\cos(x)$ 、 $\lg(x)$,对数函数 $\ln(x)$ 、 $\lg(x)$,指数函数 e^x 等。这样,就能用一条指令代替软件的一个子程序来完成函数计算。

有些计算机中还设置有多项式计算、快速傅立叶变换等相当复杂的指令。例如,用一条三地址指令:



就能完成一个多项式的计算。

$$P(X) = C(0) + C(1)X + C(2)X^2 + \dots$$

其中,arg 是变量 x 的值,degree 是多项式的阶,tbladdr 是系数表的首地址。

在事务处理应用中,经常有十进制运算。如果把十进制数先转换成二进制数,计算完成后,再转换回十进制,程序的时间开销和空间开销都会很大。因此,在许多计算机中都设置有一套十进制运算指令。也有些计算机设置有二进制到十进制和十进制到二进制的转换指令。

另外,在事务处理应用中也经常遇到码制转换问题,如 ASCII 码与 BCD 码之间的转换,ASCII 码与 EBCDIC 码之间的转换。在 IBM 370 计算机中就设置有专门的码制转换指令,用一条指令就能完成一个字串的码制转换。

3. 增强程序控制指令的功能

上一节中介绍了两种基本的程序控制指令:转移指令和子程序控制指令。在一般计算机系统中,有了这两类指令就能编写任何程序了。

由于循环在一般程序占有相当大的比例,而且,许多循环程序中的循环体本身往往很短,在一般高级语言中,循环体中只有一条语句的约占 40%左右,有一至三条语句的约占 70%左右。因此,循环控制指令在整个循环程序中占据了相当大的比例。

一般循环程序的结构如图 2.23 所示,其中虚线框内的功能通常要用三条指令来完成,一条加法指令、一条比较指令和一条条件转移指令。为了支持循环程序的快速执行,缩短循环程序的代码长度,可以用一条循环控制指令来实现这些功能。在 IBM370 计算机中,一条“大于转移”指令就是为了实现这一功能而设置的。指令格式如下:

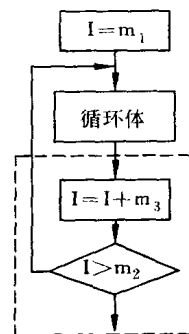


图 2.23 循环程序结构

8	4	4	4	12
OPC	R1	R3	B2	D2

其中,在第一地址码给出的通用寄存器(R1)中存放循环变量 I,在第三地址码给出的通用寄存器(R3)中存放循环结束变量 m2,循环的步长 m3 隐含存放在比第二地址码给出的通用寄存器编号大 1 的那个通用寄存器(R3+1)中。转移地址由第二地址码(B2)+D2 形成。类似的指令还有多条,如:大于等于转移、小于转移、小于等于转移等。另外,IBM370 计算机中还有一类称为“计数转移”的程序控制指令,它们的指令格式和指令功能与“大于转移”等指令相同。这类指令用于循环步长为 1 的循环程序,由于步长隐含为 1,因此,不需要 R3+1 通用寄存器。

2.4.2.2 对高级语言和编译程序的支持

大多数人都已经习惯用高级语言编写程序,只有在极少数有特殊要求的场合才用机器语言或汇编语言编写程序。目前在机器上实际运行的绝大多数程序都是用高级语言编写、并经编译程序编译后生成的目标程序。

然而,大多数高级语言与一般计算机的机器语言的语义差距非常大,通常用高级语言编写的程序经编译程序编译后生成的目标程序,与直接用机器语言或汇编语言编写的程序相比,时间开销和空间开销都要大一个数量级。

因此,改进指令系统,增加对高级语言和编译程序的支持,缩小高级语言与机器语言的差距,就能提高整个计算机系统的性能。

面向高级语言和编译程序增强指令系统的途径主要有两个:

1. 增强对高级语言和编译程序支持的指令的功能

在用高级语言编写的源程序中,对各种语句的使用频度和执行时间进行统计和分析。对使用频度高、执行时间长的语句,增强有关指令的功能,或增加相关的专门指令,从而达到缩短目标程序长度、减少目标程序执行时间的目的。这样做,同时也能缩短编译所用的时间。

例如,FORTRAN 语言和 COBOL 语言中各种主要语句的使用频度如表 2.23 所示。

表 2.23 高级语言主要语句使用频度统计

语 言	一元赋值	其它赋值	IF	GOTO	I/O	DO	CALL	其它
FORTRAN	31.0	15.0	11.5	10.5	6.5	4.5	6.0	15.0
COBOL	42.1	7.5	19.1	19.1	8.46	0.17	0.17	3.4

从表 2.23 中看到,一元赋值在高级语言中所占比例最大,由于实际上它是用数据传送指令来实现的,因此,增强数据传送指令的功能,缩短这类指令的执行时间是对高级语言非常有力的支持。

在其它赋值语句中,增 1 操作所占的比例比较大,因此,许多机器都设置有专门的增 1 指令。

条件转移(IF)和无条件转移(GOTO)语句所占的比例达到 22%和 38.2%,因此,增强转移指令的功能,增加转移指令的种类是必要的。另外,一般高级语言中都没有条件码,而大多数机器指令都生成条件码,因此,取消一般指令中的条件码,改由条件转移指令本身生成条件码是对高级语言的支持。

另外,前面已经介绍过,增强体系结构的规整性,减少体系结构中各种例外情况,是对编译程序的有力支持。

2. 高级语言计算机

缩小高级语言与机器语言的差距时,如果走到极端就是把高级语言与机器语言合二为一,即所谓的高级语言计算机。在这种机器中,高级语言不需要经过编译,直接由机器的硬件来执行,如 LISP 计算机、PROLOG 计算机等

针对多种高级语言,可以研制各种 VLSI 芯片,在同一台机器上可以安装多种高级语言的专用芯片。也可以采用微程序技术,通过微程序存储器的动态加载来实现在同一台机器上具有多种高级语言。

2.4.2.3 操作系统的优化实现

任何一种计算机系统都必须有操作系统的支撑才能工作,而操作系统又必须用指令系统来实现。指令系统对操作系统的支持主要有:

处理机工作状态和访问方式的转换;

进程的管理和切换;

存储管理和信息保护;

进程的同步和互斥,信号灯的管理等。

支持操作系统的有些指令属于特权指令,对一般用户是不公开的。

尽管有些指令的使用频度很低,但是,如果没有这些指令的支持,操作系统将很难实现,或根本不能实现,如处理机状态的转换、进程的切换、信号灯的管理等方面所使用的有关指令。

2.4.3 精简指令系统(RISC)

精简指令系统计算机(RISC)是 80 年代提出的一种新的设计思想。目前运行中的许多处理机都采用了 RISC 体系结构,如 SUN 公司的 SPARC、SuperSPARC、UltraSPARC, SGI 公司的 R4000、R5000、R10000, IBM 公司的 Power、Power PC, Intel 公司的 80860、80960, DEC 公司的 Alpha, Motorola 公司的 88100, HP 公司的 HP3000/930 系列、950 系列等。另外,在有些典型的 CISC 处理机中也采用了 RISC 设计思想,如 Intel 公司的 80486、Pentium、Pentium Pro、Pentium I 等。

2.4.3.1 从 CISC 到 RISC

从计算机出现,特别是 1964 年 IBM360 系列计算机推出之后,人们一直在改进计算机的结构,不断增强指令系统功能。上一节中已经介绍了面向目标程序,面向高级语言和编译程序,面向操作系统的优化实现等方面来增强指令的功能。到了 70 年代,许多典型计

算机的指令系统已经非常庞大,指令的功能相当复杂。表 2.24 列出了当时四种典型计算机的结构特点。

从表 2.24 中看到,当时计算机的结构已经非常复杂。指令种类很多,寻址方式复杂,有大量的访问主存储器的指令,许多复杂指令的实现不得不借助于微程序,从而造成微程序容量大幅度增加。

表 2.24 四种典型 CISC 处理机的结构特点

机 型 (生产年代)	IBM370/168 (1973)	VAX-11 (1978)	iAPX 432 (1982)	Dorado (1978)
指令种类	208	303	222	270
微程序容量	420K	480K	64K	136K
指令长度	16—48	16—456	6—321	8—24
采用的工艺	ECL MSI	TTL MSI	NMOS VLSI	ECL MSI
指令操作类型	存储器-存储器 存储器-寄存器 寄存器-寄存器	存储器-存储器 存储器-寄存器 寄存器-寄存器	面向堆栈 存储器-存储器	面向堆栈
Cache 容量	64KB	64KB	0	64KB

1975 年,IBM 公司率先组织力量,开始研究指令系统的合理性问题。在 John Coche 领导下,于 1979 年研制出一种用于电话交换系统的 32 位小型计算机 IBM 801,它有 120 条指令,工作速度 10MIPS,这是世界上第一台采用 RISC 思想的计算机系统。1986 年,IBM 正式推出采用 RISC 体系结构的工作站 IBM RT PC,并采用了新的虚拟存储技术,主要用来完成 CAE、CAD、CAM 等方面的任务。

从 1979 开始,美国加州伯克利分校以 David Patterson 为首的研究小组开展了这方面的研究工作,他们指出 CISC 存在有多方面的缺点。归纳起来,CISC 指令系统主要存在如下三方面的问题:

1. 20 %与 80 %规律

CISC 中,各种指令的使用频度相差很悬殊,大量的统计数字表明,大约有 20%的指令使用频度比较高,占据了 80%的处理机时间。换句话说,有 80%的指令只在 20%的处理机运行时间内才被用到。

我们在典型的 CISC 处理机 IBM PC 上对 Intel 8088 处理机的指令系统进行了分析研究。在 C 语言编译程序和 PROLOG 解释程序中,各种指令的使用频度和执行时间的分布情况如表 2.25 所示。其中,指令使用频度是指某一种指令在整个程序存储空间中所占的比例,它是静态的。指令执行时间是指某一种指令在整个程序的运行过程中所占的时间比例,这是动态的。

8088 处理机的指令种类大约有 100 种。从表 2.25 中看到,前 11 种指令的使用频度、前 8 种指令的运行时间就已经超过了 80%,前 20 种(20%)指令的使用频度达到 91.1%,运行时间达到 97.72%,也就是说,其余 80%指令的使用频度只有 8.9%,只占 2.28%的

处理机运行时间。

表 2.25 Intel 8088 处理机指令系统使用频度和执行时间统计

指令使用频度				指令执行时间			
序 号	指令名称	占百分比	累计百分比	序 号	指令名称	占百分比	累计百分比
1	MOV	24.85	24.85	1	IMUL	19.55	19.55
2	PUSH	10.36	35.21	2	MOV	17.44	36.99
3	CMP	10.28	45.49	3	PUSH	11.11	48.10
4	JMPcc	9.03	54.52	4	JMPcc	10.55	58.65
5	ADD	6.80	61.32	5	CMP	7.80	66.45
6	POP	4.14	65.46	6	CALL	7.27	73.72
7	RET	3.92	69.38	7	RET	4.85	78.57
8	CALL	3.89	73.27	8	ADD	3.27	81.84
9	JUMP	2.70	75.97	9	JMP	3.26	85.10
10	SUB	2.43	78.40	10	LES	2.83	87.93
11	INC	2.37	80.77	11	POP	2.61	90.54
12	LES	1.98	82.75	12	DEC	1.49	92.03
13	REPN	1.92	84.67	13	SUB	1.18	93.21
14	IMUL	1.69	86.36	14	XOR	1.04	94.25
15	DEC	1.37	87.73	15	INC	0.99	95.24
16	XOR	1.13	88.86	16	LOOPcc	0.64	95.88
17	REP NZ	0.78	89.64	17	LDS	0.64	96.52
18	CLD	0.54	90.18	18	CMPS	0.44	96.96
19	LOOPcc	0.52	90.70	19	MOVS	0.39	97.35
20	TEST	0.40	91.10	20	JCXZ	0.37	97.72
21	SYI	0.40	91.50	21	LODS	0.31	98.03
22	LDS	0.39	91.89	22	REPN	0.28	98.31
23	LODS	0.35	92.24	23	INTec	0.26	98.57
24	AND	0.35	92.59	24	STOS	0.25	98.82
25	SHR	0.33	92.92	25	SHR	0.23	99.05
26	STOS	0.31	93.23	26	LEA	0.12	99.17
27	MOVS	0.29	93.52	27	OR	0.11	99.28
28	JCXZ	0.29	93.81	28	REP NZ	0.11	99.39
29	SH/AL	0.27	94.08	29	AND	0.09	99.48
30	CMPS	0.27	94.35	30	TEST	0.09	99.57

如果把 Intel 8088 处理机的所有指令分为数据传送类指令、算术运算类指令、逻辑运算/位操作类指令、字符串处理类指令、程序控制类指令和处理机控制类指令等 6 类,我们用 7 个测试程序进行统计,统计结果见表 2.26。从表中发现,只有三类指令的使用频度比较高,均占 90%,其余三类指令的使用频度均不到 10%。这三类指令是:数据传送类指令、算术运算类指令和程序控制类指令。

表 2.26 Intel 8088 处理机各类指令使用频度统计

指令类型	8 种应用程序							平均值
	F1	F2	F3	F4	F5	F6	F7	
数据传送	34.25	35.85	28.84	20.12	25.04	24.33	34.31	30.25
算术运算	24.97	22.34	45.32	43.65	45.72	45.42	28.28	36.24
逻辑运算/位操作	3.40	4.34	7.63	7.49	6.38	3.97	4.89	5.44
字符串处理	2.42	4.22	2.72	2.01	2.10	2.35	2.10	2.86
转移指令	34.84	32.99	15.34	17.63	20.52	25.72	30.29	25.33
处理器控制	0.13	0.26	0.15	0.10	0.24	0.19	0.14	0.19

2. VLSI 技术的发展引起的问题

进入 80 年代后, VLSI 技术的发展非常迅速, 往往每 3 至 4 年集成度就提高一个数量级。VLSI 工艺要求规整性, 而 CISC 处理机中, 为了实现大量的复杂指令, 控制逻辑极不规整, 给 VLSI 工艺造成很大困难。而 RISC 处理机的控制逻辑非常简单; 它所需要的大量的通用寄存器等是非常规整的, 正好适应了 VLSI 工艺的要求。

在 CISC 处理机中, 大量使用微程序技术以实现复杂的指令系统。70 年代之前, 一般用磁芯做主存储器, 用半导体做控制存储器, 两者的速度相差 5 到 10 倍。从 70 年代后期开始, 大量使用 DRAM(动态随机存储器)做主存储器, 使得主存与控存的速度相当, 从而使许多简单指令没有必要用微程序来实现, 而复杂的指令, 用微程序实现和用简单指令组成的子程序实现已经没有多大区别。

由于 VLSI 的集成度迅速提高, 使得生产单芯片处理机成为可能。在单芯片处理机内, 希望采用规整的硬布线控制逻辑, 不希望用微程序。

3. 软硬件的功能分配问题

在 CISC 中, 为了支持目标程序的优化, 支持高级语言和编译程序, 增加了许多复杂的指令, 用一条指令来代替一串指令。这些复杂指令简化了目标程序, 缩小了高级语言与机器指令之间的语义差距。然而, 增加了这些复杂指令, 是否能缩短程序的执行时间呢? 实际上, 往往不是这样。

为了实现复杂的指令, 不仅增加了硬件的复杂程度, 而且使指令的执行周期大大加长。例如, 为了支持编译程序的对称性要求, 一般的运算型指令都能直接访问主存储器, 从而使指令的执行周期数增加, 数据的重复利用率降低。据统计, 一般 CISC 处理机的指令平均执行周期都在 4 以上, 有些在 10 以上, 如 Intel 公司的 8088, Motorola 公司的 MC68010, DEC 公司的 MICRO VAX-11 等。

这里有一个软件与硬件的功能如何恰当分配的问题。在 CISC 中, 通过增强指令系统的功能, 简化了软件, 增加了硬件的复杂程度。然而, 由于指令复杂了, 指令的执行时间必然加长, 从而有可能使整个程序的执行时间反而增加。因此, 在计算机体系结构设计中, 软硬件的功能分配必须恰当。

1981 年, Patterson 等人研制成功了 32 位的 RISC I 微处理器。总共 31 种指令(算术逻辑指令 12 种, 访问存储器指令 8 种, 程序控制指令 7 种, 其它指令 4 种), 3 种数据类型, 只有变址寻址和相对寻址两种寻址方式。按字节编址, 指令采用三地址, 有少量二地址

和一地址指令,指令字长都是 32 位。时钟频率为 8MHz,所有指令都在一个周期(500ns)内完成。只有 LOAD/STORE 这里可以访问存储器,其它指令的操作都在通用寄存器之间进行,有 78 个通用寄存器,采用寄存器窗口技术。用 NMOS VLSI 实现。该处理器的设计错误和布线错误只有各 12 个,而 MC68000 各有 70 个,Z8000 为 60 个和 100 个。控制部分的芯片面积只占约 6%,而 MC68000 为 50%,Z8000 为 53%。研制周期只用了 10 个月。其性能比当时最先进的商品化微处理器 MC68000 和 Z8002 快 3 至 4 倍,有些方面超过了 PDP-11/70 和 VAX-11/780 小型机。

1983 年,他们又研制出 RISC II,指令种类扩充到 39 种(增加了采用变址寻址方式的取数指令 5 种和存数指令 3 种),使用单一的变址寻址方式。通用寄存器增加到 138 个。仍采用 NMOS 工艺。时钟频率提高到 12MHz,指令执行周期缩短为 330ns。控制部分只占总芯片面积的 10%。该处理器的设计错误只有约 18 个,布线错误只有约 12 个。

目前,RISC 思想已经被人们普遍接受,许多 CPU 采用了 RISC 结构,一些典型的 CISC 处理机也吸收了 RISC 设计思想。

2.4.3.2 RISC 的定义与特点

RISC 是一种计算机体系结构的设计思想,它不是一种产品。RISC 是近代计算机体系结构发展史中的一个里程碑。然而,直到现在,RISC 还没有一个确切的定义。这里推荐一个由卡内基梅隆(Carnegie Mellon)大学的教师在一篇论文中提出的关于 RISC 的定义。尽管这个定义并不完整,但还是明确地勾划出了 RISC 思想的一些主要特点。

卡内基梅隆大学是这样论述 RISC 特点的:

1. 大多数指令在单周期内完成。指令系统中的大多数指令只执行一个简单的和基本的功能,这些指令可以比较快地在单个周期内执行完成,并减少指令的译码和解释所需要的开销。

2. 采用 LOAD/STORE 结构。因为访问存储器指令需要的时间比较长,因此,在指令系统中要尽量减少这类指令,只保留不可再少的 LOAD 和 STORE 两种访问存储器的指令。LOAD/STORE 结构的另一种理解方法是:凡是在 CPU 执行部件中所需要的操作数都来自于通用寄存器中,运算结果也只放到通用寄存器中。LOAD/STORE 结构也有助于实现大多数指令在单周期内完成。

3. 硬布线控制逻辑。硬布线控制逻辑可以使大多数指令在单周期内执行完成,减少了微程序技术中的指令解释开销。

4. 减少指令和寻址方式的种类。这一特点也可以简化控制部件的结构,加快指令的执行速度。

5. 固定的指令格式。该特点可以使指令的译码逻辑电路简化,从而使控制部件的速度加快。

6. 注重译码的优化。

这个定义有一定的局限性,不太完整,而且,随着计算机技术的不断发展,RISC 思想也在发展中。例如,一些新出现的 RISC 处理机的控制部件,除了采用硬布线逻辑之外,也采用了微程序技术。有些 RISC 处理机的指令种类并不太少,或者说,并不比 CISC 处理机

的指令种类少。

从目前的发展来看,RISC 体系结构还应具有如下特点:

1. 面向寄存器结构。
2. 十分重视提高流水线的执行效率。要提高 RISC 处理机的速度,必须采用流水线,而且,要尽量减少断流,提高流水线的效率。
3. 重视优化编译技术。优化编译技术在提高系统性能中发挥很重要的作用,改变了过去认为提高计算机速度仅仅依靠硬件的传统观点。

因此,高效率的流水线和优化编译技术是现代 RISC 系统必须十分注重的两点。这比卡内基梅隆大学的定义更加全面了。

90 年代初,IEEE 的 Michael Slater 对于 RISC 的定义做了如下描述:

RISC 处理器所设计的指令系统应使流水线处理能高效率执行,并使优化编译器能生成优化代码。

1. RISC 为使流水线高效率执行,应具有下述特征:

- (1) 简单而统一格式的指令译码;
- (2) 大部分指令可以单周期执行完成;
- (3) 只有 LOAD 和 STORE 指令可以访问存储器;
- (4) 简单的寻址方式;
- (5) 采用延迟转移技术;
- (6) 采用 LOAD 延迟技术。

2. RISC 为使优化编译器便于生成优化代码,应具有下述特征:

- (1) 三地址指令格式;
- (2) 较多的寄存器;
- (3) 对称的指令格式。

2.4.3.3 减少指令平均执行周期数是 RISC 思想的精华

读者往往会提出这样一个问题:精简指令系统计算机 RISC 的指令系统精简了,复杂指令系统计算机 CISC 的一条指令,在 RISC 中要用一串指令才能实现,那么,为什么 RISC 执行程序的速度比 CISC 还要快呢?

这里,有一个很简单,也是很重要的公式。任何一个程序在计算机上的执行时间可以用下面的公式来计算:

$$P = I \cdot CPI \cdot T$$

其中:

- P 是执行这个程序所使用的总的时间;
- I 是这个程序所需执行的总的指令条数;
- CPI 是每条指令执行的平均周期数;
- T 是一个周期的时间长度。

表 2.27 列出了 CISC 与 RISC 的三个参数 I 、 CPI 和 T 的比较情况。从这三个参数的比较中可以得出如下结论:

表 2.27 CISC 与 RISC 的 I 、 CPI 和 T 的比较

类 型	指令条数 I	指令平均周期数 CPI	周期时间 T
CISC	1	2~15	33ns~5ns
RISC	1.3~1.4	1.1~1.4	10ns~2ns

1. 对于程序所执行的总的指令条数 I

由于 RISC 的指令都比较简单, CISC 中的一条复杂指令所完成的功能在 RISC 中可能要用几条指令才能实现。对于同一个源程序, 分别编译后生成的动态目标代码, 显然 RISC 的要比 CISC 的多。但是, 由于 CISC 中复杂指令使用的频度很低, 程序中使用的绝大多数指令都是与 RISC 一样的简单指令, 因此, 实际上的统计结果表明, RISC 的 I 长度只比 CISC 的长 30% 至 40%。

2. 对于指令平均执行周期数 CPI

由于 CISC 一般是用微程序实现的, 一条指令往往要用好几个周期才能完成, 一些复杂指令所要的周期数就更多。根据统计, 大多数 CISC 处理机中指令平均执行周期数 CPI 在 4 到 6 之间。而 RISC 的大多数指令都是单周期执行的, 它们的 CPI 应该是 1, 但是, 由于 RISC 中还有 LOAD 和 STORE 指令, 也还有少数复杂指令, 所以, CPI 要略大于 1。据报道, SUN 公司的 SPARC 处理机的 CPI 为 1.3 到 1.4, SGI 公司的 MIPS 处理机的 CPI 为 1.1 到 1.2。

3. 对于一个周期的时间长度 T

由于 RISC 一般采用硬布线逻辑实现, 指令要实现的功能都比较简单, 所以, RISC 的 T 通常要比 CISC 的 T 小。从报道中也可以看到, 目前使用中 RISC 处理机的工作主频一般要比 CISC 处理机高。

从表 2.27 中可以很快计算出, RISC 的速度要比 CISC 快 3 倍左右。其中的关键在于 RISC 的指令平均执行周期数 CPI 减小了, 这正是 RISC 设计思想的精华。

减小 CPI 是多个方面共同努力的结果。在硬件方面, 采用硬布线控制逻辑, 减少指令和寻址方式的种类, 使用固定的指令格式, 采用 LOAD/STORE 结构, 指令执行过程中设置多级流水线等, 软件方面十分强调优化编译技术的作用。

当然, RISC 设计思想也可以用于 CISC 中。例如, Intel 公司的 80x86 处理机的指令平均执行周期数在不断缩小, 8088 的指令平均执行周期数大于 20, 80286 的指令平均执行周期数大约是 5.5, 到了 80386, 指令平均执行周期数进一步减小到 4 左右, 而 80486 处理机的指令平均执行周期数已经接近 2。

目前, 微处理器的工作主频已经达到几百兆赫, 已经接近于所用半导体器件的极限工作主频。因此, 将来提高处理机速度的主要技术途径仍然是减少指令平均执行周期数。采用超标量、超流水线、VLIW (超长指令字) 体系结构, 可以使指令的平均执行周期数小于 1, 即平均每个周期执行超过 1 条指令。目前已经商品化的微处理机, 其内部大都有 4 个至 16 个功能部件并行工作, 每个周期可以平均执行 2 条以上指令。

2.4.3.4 RISC 的关键技术

RISC 要达到很高的性能,必须有相应的技术支持。目前,在 RISC 处理机中主要采用如下几种技术:

1. 延时转移技术

在 RISC 处理机中,指令一般采用流水线方式工作。取指令和执行指令并行进行。如果取指令和执行指令各需要一个周期,那么,在正常情况下,每一个周期就能执行完一条指令。然而,在遇到转移指令时,流水线就可能断流。如图 2.24(a)所示的一个简单程序,当执行 JMP NEXT2 指令时,由于转移的目的地址要在指令执行完成后才能产生,这时,下一条指令已经取出来了,因此,必须把已经取出来了的指令 3 作废,并按照转移地址重新取出正确的指令,如图 2.24(b)所示。如果已经取出来的指令 3 不作废,而继续执行,那么,整个程序的语义就可能发生错误。有两种办法可以作废指令 3:一种是用硬件,即控制指令 3 的执行结果不写入目的寄存器;另一种办法是通过软件在转移指令后面加入一条空操作指令(NOP)。但是,无论采用那一种办法,都要浪费一个周期。

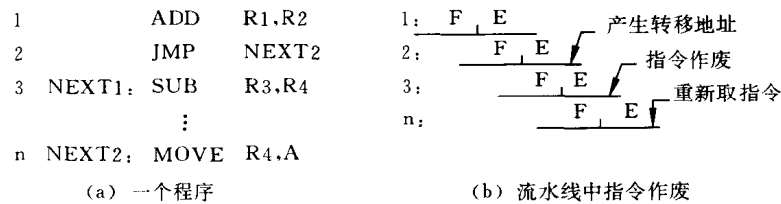


图 2.24 因转移指令引起的流水线断流

如果把 JMP NEXT2 这条指令提前执行,情况就完全不同了。如图 2.25(a)所示,把第一和第二条指令交换位置。程序在流水线中执行的情况如图 2.25(b)所示。这时,流水线没有断流情况发生,程序语义也正确。

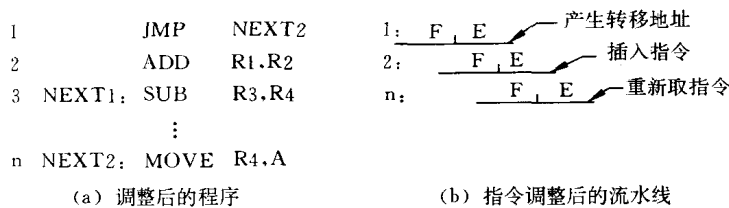


图 2.25 采用延时转移技术的指令流水线

从图 2.25 中看,也可以认为是在转移指令之后插入了一条有效的指令,而转移指令好像被延迟执行了,因此,把这种技术称为延迟转移技术。

采用指令延迟转移技术时,指令序列的调整由编译器自动进行,一般不需要人来干预。但是,如果要在目标程序一级调试程序,这种已经被调整过了的程序将很难看懂,很容易引起人们的误解。

很容易会提出这样一个问题:如果是条件转移指令时,还能不能采用延迟转移技术呢?我们来看下面的程序:

```

1:      MOVE  R1, R2
2:      CMP   R3, R4      ;(R3)与(R4)比较
3:      BEQ   EXIT        ;如果(R3)=(R4)则转移到 NEXT
4:      ADD   R4, R5
      .....
N: NEXT: MOVE  R4, A

```

重新调整一下程序的指令序列,把原来的第一条指令插入到条件转移指令之后,得到一个新的程序:

```

1:      CMP   R3, R4      ;(R3)与(R4)比较
2:      BEQ   EXIT        ;如果(R3)=(R4)则转移到 NEXT
3:      MOVE  R1, R2      ;插入一条指令,但不能有数据相关,不能改变条件码
4:      ADD   R4, R5
      .....
N: NEXT: MOVE  R4, A

```

以上程序在执行时流水线就不会有断流情况发生。然而,必须注意:调整指令序列时一定不能改变原来程序的数据相关关系,即被移动指令中的所有数据存储单元与移动过程中所经过的指令的所有数据存储单元之间不能有数据的读-写、写-读和写-写相关。另外,还要求被移动的指令不要破坏机器的条件码,至少不要影响后面的条件码测试指令所要求的条件码。

如果找不到符合上述条件的指令来调整程序中的指令序列,那么,编译程序必须在条件转移指令后面插入一条空操作指令。如果指令的执行过程分为多个流水段,则要插入多条空操作指令。

2. 指令取消技术

采用指令延时技术,遇到条件转移指令时,调整指令序列非常困难,在许多情况下找不到可以用来调整的指令。有些 RISC 处理机采用指令取消技术。

在使用指令取消技术的处理机中,所有转移指令和数据变换指令都可以决定下面待执行的指令是否应该取消。如果指令被取消,其效果相当于执行了一条空操作指令,不影响程序的运行环境。

为了提高程序的执行效率,应该尽量少取消指令,以保持指令流水线处于充满状态。因此,可以采用如下规则:如果是向后转移(转移的目标地址小于当前程序计数器 PC 的值),则在转移不成功时取消下条指令,否则,执行下条指令;如果是向前转移,则正好相反,在转移不成功时执行下条指令,否则,取消下条指令。

下面看两个具体的例子。首先是一个向后转移的例子。调整前的程序如图 2.26(a)所示,调整后的程序如图 2.26(b)所示。

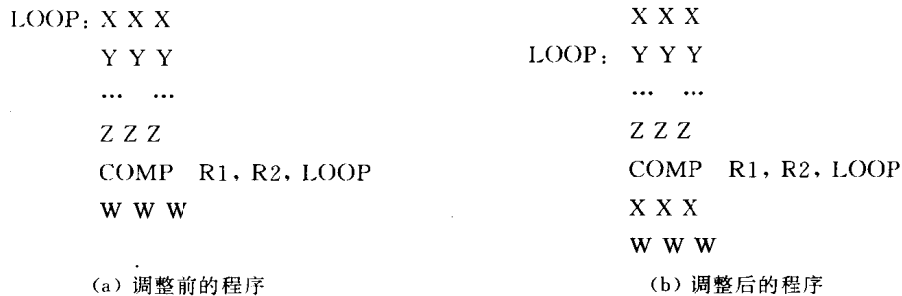


图 2.26 指令取消技术

循环体的第一条指令(X X X)经调整后安排在两个位置,第一个位置是在循环体的前面,即在进入循环之前要先执行一次。第二个位置安排在循环体的后面,即在循环的出口条件判断指令 COMP 的下面。如果转移成功,则执行下面的 X X X 指令,然后返回到 LOOP;如果转移不成功,则取消下面的 X X X 指令,接着执行 W W W 指令。

由于向后转移时,绝大多数情况下是转移成功的,只有在循环全部结束的最后一次,转移才不成功。因此,采用这种指令取消技术能够使指令流水线在绝大多数情况下不断流,保持很高的流水效率。

对于向前转移的情况,即 IF THEN 控制结构,有如下一个程序:

```

R R R ]
... ..]  “IF”部分的程序代码
S S S ]
COM R1, R2, THRU
T T T ]
... ..]  “THEN”部分的程序代码
U U U ]
THRU: V V V
    
```

如果 COMP 指令的转移条件不成立,则下条指令 T T T 不取消,“THEN”部分的程序代码照常执行;如果转移条件成立,下条指令 T T T 的执行被取消,程序执行转向 THRU 位置,“THEN”部分的程序代码全部不执行。

由于向前转移成功与不成功的概率通常各为 50%,因此,采用正常的指令取消技术就可以了。

另外,对于条件分支中只有一条指令的情况,可以采用隐含转移技术。例如,为了实现下面的语句:

IF (a<b) THEN b=b+1

用汇编语言可以写成:

```

COMP >=, Ra, Rb    ;a 与 b 已经存放在通用寄存器 Ra 和 Rb 中
INC Rb
    
```

指令 COMP 比较 a 与 b 的大小,若(Ra)>=(Rb),则取消下条指令,否则,执行下条

指令。

3. 重叠寄存器窗口技术

RISC 的指令系统比较简单, CISC 中的一条复杂指令, 在 RISC 中通常要用一段子程序来实现。因此, RISC 程序中的 CALL 和 RETURN 指令要比 CISC 程序中的多。

在执行 CALL 指令时, 必须把硬件现场(主要包括程序计数器和处理机状态字)和程序本身的软件现场(主要指在子程序中要使用的通用寄存器等)保存到主存储器中。另外, 还要把执行子程序所需要的参数从主程序传送过去。在执行 RETURN 指令时, 要做相反的工作, 最后把运算结果传送回主程序。因此, 执行 CALL 和 RETURN 指令时, 访问存储器的信息量非常大。据统计, 在 PASCAL 语言和 C 语言中分别有 15% 和 12% 的 CALL 和 RETURN 操作, 而它们访问存储器的信息量却占整个访存信息量的 44% 和 45%。

为了使 CALL 和 RETURN 操作尽量少访问存储器, 美国加州大学伯克利分校的 F. Baskett 提出重叠寄存器窗口 (overlapping register window) 技术, 并且首先在 RISC I 上应用。在 RISC II 上, 寄存器的数量增加到 138 个。目前, 重叠寄存器窗口技术已经成为 RISC 的一种基本技术。

重叠寄存器窗口的基本思想是: 在处理机中设置一个数量比较大的寄存器堆, 并把它划分成很多个窗口。每个过程使用其中相邻的三个窗口和一个公共的窗口, 而在这些窗口中有一个窗口是与前一个过程共用, 还有一个窗口是与下一个过程共用的。与前一过程共用的窗口可以用来存放前一过程传送给本过程的参数, 同时也存放本过程传送给前一过程的计算结果。同样, 与下一过程共用的窗口可以用来存放本过程传送给下一过程的参数和存放下一过程传送给本过程的计算结果。

图 2.27 是 RISC II 中采用的重叠寄存器窗口。共有 138 个寄存器, 分成 17 个窗口, 其中, 有一个由 10 个寄存器组成的窗口是全局窗口, 能被所有过程访问。另外有 8 个窗口, 每个窗口各 10 个寄存器, 分别作为 8 个过程的局部寄存器。还有 8 个窗口, 每个窗口各有 6 个寄存器, 是相邻两个过程公用的, 称为重叠寄存器窗口。每个过程均可以访问 32 个寄存器, 其中, 有 10 个是所有过程公用的全局寄存器, 有 10 个是只供本过程使用的局部寄存器, 有 6 个是与上一过程公用的寄存器, 还有 6 个是与下一过程公用的寄存器。

只要调用的深度不超过规定的层数(如 8 层), 重叠寄存器窗口技术可以减少大量的访存操作。当调用层数超过规定层数时, 称为寄存器溢出, 这时, 可以在主存中开辟一个堆栈, 把超过规定层数的寄存器中的内容压入堆栈中。

在 SUN 公司的 SPARC 处理机中, 以及后来的 Super SPARC 和 Ultra SPARC 处理机中, 还把最后一个过程的公用寄存器与第一个过程的公用寄存器重叠起来, 形成一个循环圈。在调用层数很多时, 可以循环使用。

F. Baskett 等人使用 Quicksort 和 Puzzle 两个程序对寄存器窗口技术的有效性进行了测试。Quicksort 程序的特点是过程调用的次数在整个程序中所占的比例比较大, 但调用的深度不大, 而 Puzzle 程序正好相反。RISC II 与 VAX-11 两种机器的比较结果如表 2.28 所示, RISC II 的访存次数主要是寄存器窗口溢出引起的, 而 VAX-11 访存次数是为了保持和恢复通用寄存器中的内容而引起的。从表中看出, RISC II 寄存器溢出的次数很少, 只占千分之一左右, 影响也不大。由于采用了寄存器窗口技术, 由程序调用引起的访问

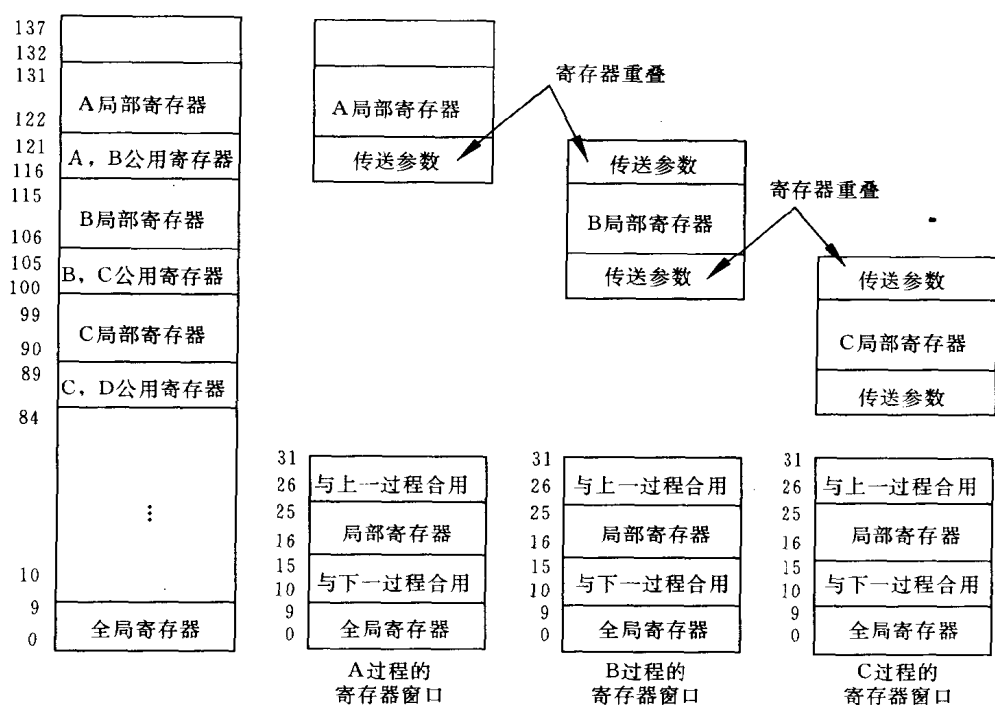


图 2.27 重叠寄存器窗口技术

存储器次数只占程序总访存次数的 1% 左右。

表 2.28 寄存器窗口技术的效果

程序名称	调用次数	最大调用深度	RISC II 溢出次数	RISC II 访存次数	VAX-11 访存次数
Quicksort	111K(0.7%)	10	64	4K(0.8%)	696K(50%)
Puzzle	43K(8.0%)	20	124	8K(1.0%)	444K(28%)

表 2.29 是 RISC II 处理机与几种 CISC 处理机的每次过程调用的开销比较。从表中可以看出,在执行时间、执行指令条数、访问存储器次数等方面,采用重叠寄存器窗口技术都是十分有效的。

表 2.29 过程调用所需开销的比较

机 器	执行指令条数	执行时间(微秒)	访问存储器次数
VAX-11	5	26	10
PDP-11	19	22	15
MC68000	9	19	12
RISC II	6	2	0.2

4. 指令流调整技术

为了使 RISC 处理机中的指令流水线高效率地工作,尽量不断流,优化编译器必须分析程序的数据流和控制流。当发现指令流有断流可能时,要调整指令序列。对有些可以通过变量重新命名来消除的数据相关,要尽量消除。这样,可以提高流水线的执行效率,缩短程序的执行时间。

例如,有图 2.28(a)这样一个简单的指令序列,由于存在 R3 寄存器的数据相关,第二条指令必须等第一条指令执行完后才能开始执行,后续的指令也是这样。如果执行一条指令需要两个机器周期,那么,每两条指令之间都要浪费一个周期。

图 2.28(b)是通过优化编译器调整后的指令序列。在两条乘法指令中用 R0 寄存器代替原来的 R3 寄存器,消除了两条乘法指令与两条加法指令之间的数据相关,并且重新调整指令序列。调整后的指令序列比原指令序列的执行速度快一倍。

ADD R1, R2, R3 ;(R1)+(R2)→R3	ADD R1, R2, R3
ADD R3, R4, R5 ;(R3)+(R4)→R5	MUL R6, R7, R0
MUL R6, R7, R3 ;(R6)×(R7)→R3	ADD R3, R4, R5
MUL R3, R8, R9 ;(R3)×(R8)→R9	MUL R0, R8, R9
(a) 调整前的指令序列	(b) 调整后的指令序列

图 2.28 指令流调整技术

调整指令序列,消除数据相关,提高流水线的工作效率还有很多种方法。在第五章流水线工作原理等章节还要详细介绍。

5. 硬件为主固件为辅

指令系统用微程序实现的主要优点是:便于实现复杂指令,便于修改指令系统,增加了机器的灵活性和适应性。主要缺点是:执行速度低。RISC 要求主要指令能在单周期内执行完成,采用微程序技术是不可能做到的。因此,RISC 必须主要采用硬连逻辑来实现指令系统。对于那些必须的复杂指令,也可用固件(微程序技术)实现。因此,目前商用的 RISC 处理机在实现指令系统时,一般都采用以硬件为主固件为辅的方法。

2.4.3.5 RISC 优化编译技术

RISC 思想在采用硬件技术提高处理机性能的同时,也十分重视软件的优化编译技术。可以说,RISC 是硬件和软件相结合的产物。没有优化编译技术的支持,RISC 处理机的性能就不可能得到充分的发挥。

RISC 的硬件设计为优化编译程序的设计带来了许多方便,同时也造成一些困难。RISC 对优化编译程序带来的方便主要有:

1. 由于 RISC 的指令系统比较简单,而且对称、均匀,优化编译程序不必为具有类似功能的指令做复杂的指令选择工作。
2. RISC 的寻址方式简单,只有 LOAD 和 STORE 指令能够访问存储器,其它指令均在通用寄存器之间进行操作。因此,可以简化优化编译器在选择寻址方式过程中要做的工作,省去了是否要生成访问存储器指令的选择工作。

3. 因为大多数指令都能在一个周期内执行完成,为优化编译器调整指令序列提供了极大的方便。

RISC 对优化编译器造成的困难主要有:

1. 优化编译器必须选择哪些变量放在通用寄存器中,哪些变量放在主存储器中,必须精心安排每一个寄存器的用法,以便充分发挥每一个通用寄存器的效率,尽量减少访问主存储器的次数。

2. 优化编译器要做数据和控制相关性分析,要调整指令的执行序列,并与硬件相配合实现指令延迟技术和指令取消技术等。

3. 要设计复杂的子程序库,因为在 CISC 中的一条指令在 RISC 中要用一段子程序来实现。所以,RISC 的子程序库通常要比 CISC 的子程序库大得多。

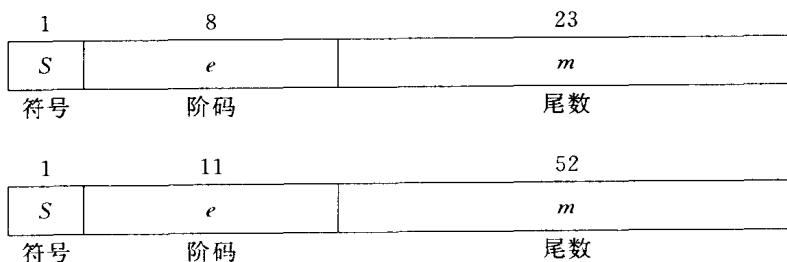
习 题 二

2.1 数据类型、数据表示和数据结构之间的关系是什么?在设计一个计算机系统时,确定数据表示的原则主要有哪几个?

2.2 尾数用补码、小数表示,阶码用移码、整数表示,尾数字长 $p=6$ (不包括符号位),阶码字长 $q=6$ (不包括符号位),尾数基值 $r_m=16$,阶码基值 $r_r=2$ 。对于规格化浮点数,用十进制表达式写出下列数据(对于前 11 项,还要写出 16 进制编码):

- | | |
|------------|--------------------|
| (1) 最大尾数; | (8) 最小正数; |
| (2) 最小正尾数; | (9) 最大负数; |
| (3) 最小尾数; | (10) 最小负数; |
| (4) 最大负尾数; | (11) 浮点零; |
| (5) 最大阶码; | (12) 表数精度; |
| (6) 最小阶码; | (13) 表数效率; |
| (7) 最大正数; | (14) 能表示的规格化浮点数个数。 |

2.3 在 IEEE 754 浮点数国际标准中,32 位单精度浮点数和 64 位双精度浮点数的格式分别如下:



对于单精度浮点数,尾数用原码、小数表示,阶码用移-127 码表示,即阶码的 0~255 分别表示阶码的真值为-127~128。尾数用 1 位符号位、23 位小数和 1 位隐藏的整数共 25 位表示。尾数的基值和阶码的基值都是 2。

当 $0 < e < 255$ 时,表示一个非零的规格化浮点数,数值为:

$$N = (-1)^s \times 2^{e-127} \times (1.m)$$

当 $e=255$, 且 $m \neq 0$ 时, 表示一个非数 NaN(Not-a-Number)。NaN 可能是在许多非确定的情况下, 如零除以零, 求负数的平方根等产生的结果。

当 $e=255$, 且 $m=0$ 时, 表示一个无穷数: $N = (-1)^s \times \infty$ 。注意 $+\infty$ 和 $-\infty$ 的表示是不同的。

当 $e=0$, 且 $m \neq 0$ 时, 表示规格化浮点数: $N = (-1)^s \times 2^{-126} \times (0.m)$ 。

当 $e=0$, 且 $m=0$ 时, 表示浮点数零: $N = (-1)^s \times 0$ 。注意 $+0$ 与 -0 的表示是不同的。

对于 64 位双精度浮点数, 阶码用移-1023 码表示, 其他规定与单精度浮点数类似。

对于 32 位单精度浮点数和 64 位双精度浮点数, 分别写出:

- (1) 最大正数; (4) 最小负数;
- (2) 最小正数; (5) 表数精度;
- (3) 最大负数; (6) 表数效率。

2.4 请证明: 在浮点数的字长和表数精度一定时, 尾数基值 r_m 取 2 或 4, 浮点数具有最大的表数范围。

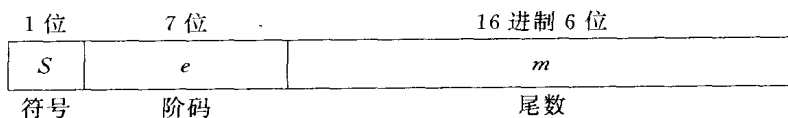
2.5 一台计算机系统要求浮点数的精度不低于 $10^{-7.2}$, 表数范围正数不小于 10^{38} , 且正、负数对称。尾数用原码、纯小数表示, 阶码用移码、整数表示。

(1) 设计这种浮点数的格式。

(2) 计算(1)所设计浮点数格式实际上能够表示的最大正数、最大负数、表数精度和表数效率。

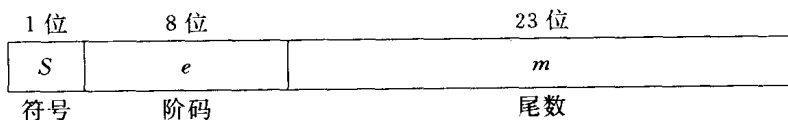
2.6 在同一种处理机内部提供有如下两种浮点数格式:

格式一: IBM 单精度浮点数标准



尾数用原码、小数表示, 阶码用移码、整数表示。尾数的基值 $r_m=16$, 长度为 6 位, 没有隐藏位。阶码的基值 $r_e=2$ 。

格式二: IEEE 754 单精度浮点数标准



尾数用原码、小数表示, 阶码用移-127 码表示, 即阶码的 0~255 分别表示阶码的真值为-127~128。尾数用 1 位符号位、23 位小数和 1 位隐藏的整数共 25 位表示。尾数的基值和阶码的基值都为 2。

(1) 把十进制数 0.2 分别表示成 IBM 单精度浮点数和 IEEE 754 单精度浮点数。

(2) 现在要把一个 IBM 单精度浮点数转成同样数值的 IEEE 754 单精度浮点数, 请

写出转换规则。

(3) 相反,如果要把 IEEE 754 单精度浮点数转成同样数值的 IBM 单精度浮点数,写出转换规则。

2.7 对于如下 4 种浮点数的舍入方法:

方法一:恒舍法

方法二:恒置法

方法三:下舍上入法

方法四: R^* 舍入法

(1) 根据舍入结果的精度不同,对这 4 种舍入方法排列次序。

(2) 根据不同舍入方法的实现难度,对这 4 种舍入方法排列次序。

(3) 根据不同的积累误差,对这 4 种舍入方法排列次序。

2.8 在浮点数运算过程中,关于警戒位的设置,请回答下列问题:

(1) 为什么要设置警戒位?

(2) 警戒位的来源有哪些?

(3) 警戒位的用处是什么?

(4) 在一个浮点处理机的运算器中,没有为任何一个寄存器专门设置警戒位,那么,这种浮点处理机采用的是什么舍入方法?

(5) 如果一个浮点处理机的累加寄存器中设置有两个警戒位,那么,这种浮点处理机采用的是什么舍入方法?

2.9 某处理机要求浮点数在正数区的积累误差不大于 2^{-p-1} ,其中, p 是浮点数的尾数长度。

(1) 选择合适的舍入方法。

(2) 确定警戒位位数。

(3) 计算在正数区的误差范围。

2.10 假设有 A 和 B 两种不同类型的处理机, A 处理机中的数据不带标志符,其指令字长和数据字长均为 32 位。B 处理机的数据带有标志符,每个数据的字长增加至 36 位,其中有 4 位是标志符,它的指令条数由最多 256 条减少至不到 64 条。如果每执行一条指令平均要访问两个操作数,每个存放在存储器中的操作数平均要被访问 8 次。对于一个由 1000 条指令组成的程序,分别计算这个程序在 A 处理机和 B 处理机中所占用的存储空间大小(包括指令和数据),从中得到什么启发?

2.11 对于一个字长为 64 位的存储器,访问这个存储器的地址按字节编址。假设存放在这个存储器中的数据中有 20% 是独立的字节数据(指与这个字节数据相邻的不是一个字节数据),有 30% 是独立的 16 位数据,有 20% 是独立的 32 位数据,另外 30% 是 64 位数据;并且规定只能从一个存储字的起始位置开始存放数据。

(1) 计算这种存储器的存储空间利用率。

(2) 给出提高存储空间利用率的方法,画出新方法的逻辑框图,并计算这种方法的存储空间利用率。

2.12 分别用变址寻址方式和间址寻址方式编写一个程序,求 $C=A+B$,其中, A 与 B 都

是由 n 个元素组成的一维数组。比较两个程序,并回答下列问题:

- (1) 从程序的复杂程度看,哪一种寻址方式更好?
- (2) 从硬件实现的代价看,哪一种寻址方式比较容易实现?
- (3) 从对向量运算的支持看,哪一种寻址方式更好?

2.13 一个处理机共有 10 条指令,各指令在程序中出现的概率如下表:

指令序号	出现的概率	Huffman 编码法	2/8 扩展编码法	3/7 扩展编码法
I_1	0.25			
I_2	0.20			
I_3	0.15			
I_4	0.10			
I_5	0.08			
I_6	0.08			
I_7	0.05			
I_8	0.04			
I_9	0.03			
I_{10}	0.02			
操作码的平均长度				
操作码的信息冗余量				

- (1) 采用最优 Huffman 编码法计算这 10 条指令的操作码最短平均长度。
- (2) 采用 Huffman 编码法编写这 10 条指令的操作码,并计算操作码的平均长度,计算与最优 Huffman 编码法相比的操作码信息冗余量。把得到的操作码编码和计算的结果填入上面的表中。
- (3) 采用 2/8 扩展编码法编写这 10 条指令的操作码,并计算操作码的平均长度,计算与最优 Huffman 编码法相比的操作码信息冗余量。把得到的操作码编码和计算的结果填入上面的表中。
- (4) 采用 3/7 扩展编码法编写这 10 条指令的操作码,并计算操作码的平均长度,计算与最优 Huffman 编码法相比的操作码信息冗余量。把得到的操作码编码和计算的结果填入上面的表中。

2.14 一台模型机共有 7 条指令,各指令的使用频度分别为 35%,25%,20%,10%,5%,3%,2%,有 8 个通用数据寄存器,2 个变址寄存器。

- (1) 要求操作码的平均长度最短,请设计操作码的编码,并计算所设计操作码的平均长度。
- (2) 设计 8 位字长的寄存器-寄存器型指令 3 条,16 位字长的寄存器-存储器型变址寻址方式指令 4 条,变址范围不小于正、负 127。请设计指令格式,并给出各

字段的长度和操作码的编码。

- 2.15 某处理机的指令字长为 16 位,有双地址指令、单地址指令和零地址指令三类,并假设每个地址字段的长度均为 6 位。

- (1) 如果双地址指令有 15 条,单地址指令和零地址指令的条数基本相同,问单地址指令和零地址指令各有多少条? 并且为这三类指令分配操作码。
- (2) 如果要求三类指令的比例大致为 1:9:9,问双地址指令、单地址指令和零地址指令各有多少条? 并且为这三类指令分配操作码。

- 2.16 为了评价各种指令系统的性能,分别在下列 5 种不同类型的处理机上计算算术表达式 $\frac{(a+b) \cdot c + d \cdot e}{f-g}$ 的值。

处理机 1: 三地址指令系统

处理机 2: 二地址指令系统

处理机 3: 一地址指令系统

处理机 4: 零地址指令系统(堆栈型处理机)

处理机 5: 二地址多累加器(通用寄存器)指令系统

- (1) 用 5 种不同类型的指令系统分别编写 5 个程序,计算上面这个算术表达式。所有程序都用直接寻址方式编写,并假设数据 a 已经存放在主存 A 单元中,其余依次类推,最终运算结果存入主存 X 单元。
- (2) 根据下面这个表的要求,对(1)编写的 5 个程序的有关数据进行统计,其中,访存次数包括取指令、到存储器中读操作数和把中间结果或最终结果写到存储器中;程序存储量是指所有指令占用的存储空间,不包括数据所占用的存储空间,因为数据所占用的存储空间对各种处理机都是相同的;访存信息量包括取指令、到存储器中读操作数和把中间结果或最终结果写到存储器中,在处理机的运算速度足够高的情况下,访存信息量实际上代表程序的执行速度。在统计程序存储量和访存信息量时,都以字节为单位,并假设一个操作码为 8 位,一个地址码为 16 位,一个数据为 32 位,一个通用寄存器地址为 4 位。

指令系统类型	指令条数	访存次数	程序存储量	访存信息量(执行速度)
三地址指令系统				
二地址指令系统				
一地址指令系统				
堆栈型指令系统				
二地址多累加器指令系统				

- (3) 根据上面表中的统计结果,对 5 种指令系统的程序存储量进行排序。
- (4) 根据上面表中的统计结果,对 5 种指令系统的程序执行速度(访存信息量)进行排序。
- (5) 根据上面的统计结果,对 5 种指令系统的性能和适用场合进行分析。

- 2.17 在一般通用计算机中,按照指令所完成的功能来划分,应该有几类指令? 各类指令的主要任务是什么?
- 2.18 在一般 RISC 处理机中,平均每个周期执行的指令条数小于 1。为了进一步提高处理机的速度,请设计两种不同的方案,使处理机平均每个周期执行的指令条数大于 1,并分析所设计方案的硬件和软件代价。
- 2.19 下面一小段程序的功能是在主存 A、B、C 三个单元中找出最大的一个数送入主存 MAX 单元中。在某 RISC 处理机中,每条指令的执行过程分为“取指令”和“执行”两个阶段,并采用两级流水线。

```

START:  LOAD   R1,  A      ;取主存 A 单元中的数据到 R1 寄存器
        LOAD   R2,  B      ;取主存 B 单元中的数据到 R2 寄存器
        LOAD   R3,  C      ;取主存 C 单元中的数据到 R3 寄存器
        CMP    R1,  R2     ;比较(R1)与(R2),即(A)-(B)
        BGE    NEXT1      ;如果(A)>(B),转向 NEXT1,否则继续执行
        MOVE   R2,  R1     ;(B)→R1
NEXT1:  CMP    R1,  R3     ;(A)-(C)
        BGE    NEXT2      ;如果(A)>(C),转向 NEXT2,否则继续执行
        MOVE   R3,  R1     ;(C)→R1
NEXT2:  STORE  R1,  MAX    ;保存(R1)到主存 MAX 单元

```

- (1) 如果在处理机中采用了指令取消技术,问上面这个程序的执行结果是否正确? 从中得到什么启示?
- (2) 如果在处理机中采用了延迟转移技术,请对上面的指令序列进行适当的调整。在确保程序语义正确的前提下,尽可能缩短程序的执行时间。
- 2.20 下面是一个数据块搬家程序。在 RISC 处理机中,为了提高指令流水线的执行效率,通常要采用指令取消技术。

```

START:  MOVE   AS,   R1      ;把源数组的起始地址送入变址寄存器 R1
        MOVE   NUM,  R2      ;把传送的数据个数送入 R2
LOOP:   MOVE   (R1),  AD-AS(R1) ;AD-AS 为地址偏移量,在程序汇编过程中
                                   计算
        INC     R1          ;增量变址寄存器
        DEC     R2          ;剩余数据个数减 1
        BGT     LOOP       ;测试 N 个数据是否传送完成
        HALT                ;停机
NUM:    N                ;需要传送的数据总数

```

- (1) 如果一条指令的执行过程分解为“取指令”和“分析”两个阶段,并采用两级流水线。为了采用指令取消技术,请修改上面的程序。
- (2) 如果 $N=100$,采用指令取消技术后,在程序执行过程中,能够节省多少个指令周期?
- (3) 如果把一条指令的执行过程分解为“取指令”、“分析”(包括译码和取操作数等)和“执行”(包括运算和写回结果等)三个阶段,并采用三级流水线。仍然要采用指令取消技术,请修改上面的程序。

第三章 存储系统

现代计算机系统都以存储器为中心,这与古典的冯·诺依曼计算机以运算器为中心不同。从程序员角度看,机器若要开始工作,必须把有关程序和数据装到存储器中之后,程序才能开始运行。

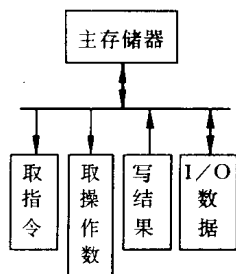


图 3.1 以存储器为中心

在程序执行过程中,中央控制器所需要的指令从存储器中取。运算器所需要的原始数据要通过程序中的访问存储器指令从存储器中取。运算结果在程序执行完成之前必须全部写到存储器中。各种输入输出设备也直接与存储器交换数据。因此,在计算机运行过程中,存储器是各种信息存储和交换的中心,如图 3.1 所示。

本章首先从系统结构的角度介绍存储系统的原理,然后介绍虚拟存储系统和 Cache 存储系统,以及最近才出现的全 all-Cache 存储系统等。

3.1 存储系统原理

存储器系统与存储器是两个完全不同的概念。如果在一台计算机中只有存储器,甚至有多种存储器,但没有存储系统,那么,这台计算机的性能将会是很差的,这些存储器的性能也不可能得到充分发挥。

一个存储器的性能通常用速度、容量、价格三个主要指标来表示。速度用存储器的访问周期(又称存储周期、存取周期、读写周期等)、读出时间、频带宽度等表示,容量用字节 B、千字节 KB、兆字节 MB 和千兆字节 GB 等单位表示,价格用单位容量的价钱表示,例如,每一个二进制位多少美分(\$C/bit)等。以下也同样用这三个指标来表示存储系统的性能。

本节主要介绍什么是存储系统(或存储体系),为什么需要存储系统,存储系统的性能指标(指容量、价格和速度)如何计算,以及如何构成存储系统等。

3.1.1 存储系统的定义

在一台计算机中,通常有多种用途不同的存储器,如主存储器(或称内存)、Cache、通用寄存器、磁盘存储器、各种缓冲存储器、磁带和光盘存储器等。从构成存储器的材料上看,有静态存储器 SRAM、动态存储器 DRAM,还有磁表面存储器和光存储器等。从存储器的访问方式看,有直接译码的、随机访问的、相联访问的,也有块交换的,甚至是手工加载的。

是否在一台计算机中,有了各种用途不同、组成材料不同、工作方式也不同的存储器

就构成了一个存储系统呢,答案是否定的。那么,究竟什么是存储系统呢?

两个或两个以上速度、容量和价格各不相同的存储器用硬件、软件、或软件与硬件相结合的方法连接起来成为一个系统。这个系统对应用程序员透明,并且,从应用程序员看它是一个存储器,这个存储器的速度接近速度最快的那个存储器,存储容量与容量最大的那个存储器相等或接近,单位容量的价格接近最便宜的那个存储器。

图 3.2 是一个典型的存储系统。它由 n 个存储器连接起来组成,这 n 个存储器的速度、容量和价格各不相同。从外部看,可以把它看作一个存储器。这个存储器的访问速度近似等于所有存储器中存储周期最小的那个,存储容量与所有存储器中容量最大的那个相等或接近,价格近似等于所有存储器中价格最便宜的那个。

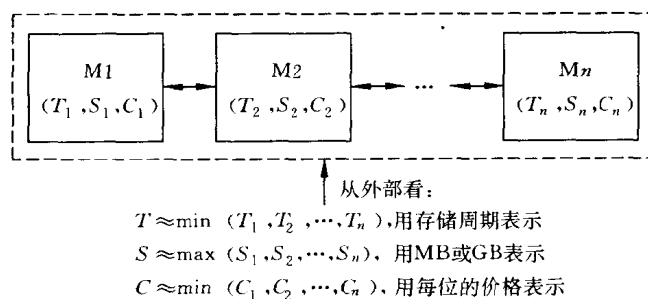


图 3.2 存储系统原理

本章主要介绍两种存储系统,一种是由 Cache 和主存储器构成的 Cache 存储系统,另一种是由主存储器和磁盘存储器构成的虚拟存储系统。在计算机系统中,这两种存储系统的作用是不相同的。Cache 存储系统的主要目标是为了提高存储器的速度,而虚拟存储系统的主要目标是为了增加存储器的存储容量。

Cache 存储系统全部用硬件来调度,因此,它不仅对应用程序员是透明的,而且对系统程序员也是透明的,它的组成原理如图 3.3 所示。

目前,Cache 一般用高速静态存储器(SRAM)实现,存储周期为几十毫微秒,存储容量在几十个千字节至几兆字节之间,价格比较贵。主存一般用动态存储器(DRAM)实现,存储周期为几百毫微秒,比 Cache 慢 5

~10 倍,存储容量在几十兆字节至几百兆字节,价格与 Cache 相比要便宜很多。这两个存储器组成存储系统之后,由于数据在 Cache 中的命中率很高,访问主存储器的绝大部分数据都能在 Cache 中访问到,因此,这个存储系统的存储周期与 Cache 非常接近。对系统程序员来说,因为只能看到主存储器,根本看不到 Cache(Cache 采用相联方式访问,对程序员来说是不编址的),因此,存储系统的容量实际上就是主存储器的容量。另外,尽管 Cache 的价格比较贵,但是,由于它在整个 Cache 存储系统所占的比例很小,因此,每位的平均价格仍然与主存储器很接近。

虚拟存储系统由主存储器与联机的外部存储器(目前一般为磁盘存储器)构成,采用

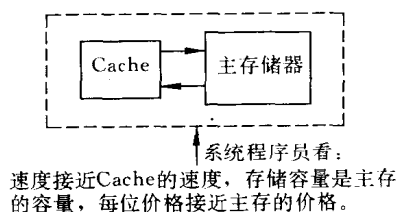


图 3.3 Cache 存储系统

硬件与软件相结合的方法来调度,如图 3.4 所示。由于虚拟存储系统需要通过操作系统的存储管理系统来调度,因此,对系统程序员来说它是不透明的,但对于在操作系统之上编程的应用程序员来说是透明的。

虚拟存储系统的访问速度与主存储器很接近,存储容量是一个很大的虚拟地址空间,这个空间的大小比主存储器的实际存储容量要大得多,也就是说,应用程序员可以在一个比主存储器的实际容量大得多的地址空间内编程。目前,许多计算机的虚拟地址空间为 4GB。由于磁盘存储器每位的价格要比主存储器便宜许多,因此,整个存储系统的每位的价格仍然接近于磁盘存储器。

表示存储系统的性能有三个主要参数:容量 S ,速度 T 和价格 C ,组成这个存储系统的每个存储器本身也有同样的三个参数,下面分析这些参数之间的关系。

为了分析方便,采用如图 3.5 所示的由两个存储器 $M1$ 和 $M2$ 组成的存储系统。两个存储器的容量、速度和价格分别为 S_1, C_1, T_1 和 S_2, C_2, T_2 ,存储系统的容量、速度和价格分别为 S, C 和 T 。

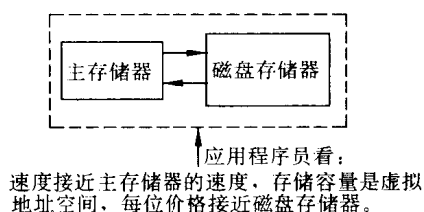


图 3.4 虚拟存储系统

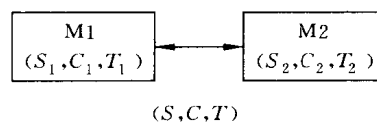


图 3.5 由两个存储器构成的存储系统

1. 存储容量 S

由于在计算机的使用者(应用程序员或系统程序员)看来,存储系统的容量要接近 $M2$ 存储器的容量,因此,可以选择 $M2$ 存储器进行编址,对于 $M1$ 存储器可以不编址,或者只在系统内部进行编址。当然,也可以面向计算机的使用者为存储系统另外设计一个抽象的地址空间,在系统内部,再对两个存储器分别进行编址,并且把两个存储器的地址映射到这个抽象的地址空间中。

对存储系统进行编址的要求是:对计算机的使用者提供尽可能大的地址空间,而且能对这个地址空间进行随机访问。

例如,对于 Cache 存储系统,面向系统程序员,选择主存储器进行编址,对 Cache 在内部采用相联访问方式进行管理。因此,从系统程序员看 Cache 存储系统,看到的是主存储器的地址空间,存储系统的容量就是主存储器的容量,即 $S = S_2$ 。

对于虚拟存储系统,由于磁盘存储器不是一种能随机访问的存储器,它的地址空间不能直接被一般指令访问,而主存储器的地址空间对于计算机的使用者来说又太小。因此,虚拟存储系统为使用者另外设计了一个虚拟地址空间,它既不是主存储器的地址空间,也不是磁盘存储器的地址空间。这个虚拟地址空间比主存储器的实际地址空间要大得多,而且采用与主存储器同样的随机访问方式。

另外,还应当指出,在一般计算机系统中,并不是整个磁盘存储器都是作为虚拟存储

系统使用的。磁盘存储器的主要用途还是用来存放各种各样的系统软件、应用软件和各种用户文件等,只有在多任务或多用户操作系统中的交换区或交换文件才是用来做虚拟存储系统的。

2. 单位容量的平均价格 C

整个存储系统的平均单位容量价格可以这样来计算:

$$C = \frac{C_1 \cdot S_1 + C_2 \cdot S_2}{S_1 + S_2}$$

当 $S_2 \gg S_1$ 时, $C \approx C_2$ 。因此,整个存储系统的单位容量价格 C 接近于比较便宜的 M2 存储器的单位容量价格 C_2 。但是, S_2 与 S_1 不能相差太大。否则,存储系统要达到比较高的性能,调度起来很困难。

3. 访问周期 T (又称存取周期、存储周期、存取时间等)

存储系统的访问周期与命中率 H 的关系非常大。命中率可以简单地定义为在 M1 存储器中访问到的概率,它一般用模拟实验的方法得到。选择一组有代表性的程序,在程序执行过程中分别统计对 M1 存储器的访问次数 N_1 和对 M2 存储器的访问次数 N_2 ,然后代入下面的(3.1)关系式计算。

$$H = \frac{N_1}{N_1 + N_2} \quad (3.1)$$

整个存储系统的访问周期可以用 M1 和 M2 两个存储器的访问周期 T_1, T_2 和命中率 H 来表示:

$$T = H \cdot T_1 + (1 - H) \cdot T_2 \quad (3.2)$$

当命中率 $H \rightarrow 1$ 时, $T \rightarrow T_1$, 即存储系统的访问周期 T 接近于速度比较快的 M1 存储器的访问周期 T_1 。

假设存储系统的访问效率为:

$$e = \frac{T_1}{T} \quad (3.3)$$

访问效率越高,说明存储系统的速度与相对比较快的那个存储器的速度就越接近,这正是我们所希望的。把(3.2)代入(3.3)得到:

$$e = \frac{T_1}{H \cdot T_1 + (1 - H) \cdot T_2} = \frac{1}{H + (1 - H) \cdot \frac{T_2}{T_1}} = f\left(H, \frac{T_2}{T_1}\right)$$

从这个关系式看到,存储系统的访问效率主要与命中率和构成存储系统的两级存储器的速度之比有关。

因此,可以得出这样的结论:如果要使存储系统的速度与相对比较快的那个存储器的速度接近,有两条途径。一条是提高命中率 H , 另一条是使构成存储系统的两个存储器的速度之比不要太低。

对于虚拟存储系统,由于两级存储器的速度相差得特别悬殊, $\frac{T_2}{T_1} > 10^5$, 如果要使访问效率 e 比较高(如 $e = 0.9$), 需要有极高的命中率 H 。

$$0.9 = \frac{1}{H + (1 - H) \cdot 10^5}$$

计算得 $H \approx 0.999\ 999$, 这么高的命中率如何达到呢?

另外, 磁盘在物理上是以块为单位(每块 512 个字节)访问的, 在逻辑上通常以 1KB 至 16KB 为单位访问。虽然磁盘存储器的寻址时间很长, 但当磁头找到要访问的数据块之后, 数据的传输速率还是相当高的。因此, 当不命中时, 通过操作系统的系统调用, 把将要使用的一大批程序和数据都调入主存储器, 使得在以后的几次乃至几十万次以上的对虚拟存储系统的访问, 都能在主存储器中命中。当然, 这要求主存储器的容量比较大, 能够一次装入比较多的程序和数据, 而且, 以前装入的程序和数据要能够比较长时间地保存下来, 并且能多次使用。这样, 尽管两级存储器的速度相差得特别悬殊, 一次不命中需要花费比较长的时间来进行调度, 然而, 由于命中率特别高, 整个虚拟存储系统的访问效率还是很高的。

对于 Cache 存储系统, 由于目前 CPU 与主存储器的速度相差两个数量级, 如果只用一级 Cache, 则要求命中率 $H \approx 0.999$, 这实际上是做不到的。通常要采用两级或三级 Cache, 再加上 CPU 内部的一些缓冲存储器, 如通用寄存器等来提高数据的重复利用率, 使得每两级之间的速度比为 5 左右。例如, 取 $T_2/T_1 = 5$ 。若要求访问效率 $e > 0.9$, 则需要命中率 $H > 0.972$ 。这实际上也很难做到。因此, 还必须采取其他措施。

实践证明, 采用预取技术可以大幅度提高命中率 H 。具体方法是: 不命中时, 在数据从主存储器中取出送往 CPU 的同时, 把主存储器相邻几个单元中的数据(称为一个数据块)都取出来送入 Cache 中。根据程序的局部性原理, CPU 以后再对 Cache 存储系统访问时, 命中率就会提高。不难理解有下面的关系式:

$$H' = \frac{H + n - 1}{n}$$

其中 n 为 Cache 的块大小与数据重复使用次数的乘积, H 是原来的命中率, H' 是采用预取技术之后的命中率。

Cache 的块大小一般在 2 至 16 个字, 预取到 Cache 中的数据的数据的重复利用率通常大于 5 次。如果取 Cache 的块大小为 4 个字, 预取到 Cache 中的数据的数据的重复利用率为 5 次, Cache 存储系统原来的命中率为 $H = 0.8$, 则有: $n = 4 \times 5 = 20$, 采用预取技术之后, 命中率提高到:

$$H' = \frac{H + n - 1}{n} = \frac{0.8 + 20 - 1}{20} = 0.99$$

这时, 如果构成 Cache 存储系统的两级存储器的速度之比为 5, 则这个 Cache 存储系统的访问效率 $e = 0.96$ 。

3.1.2 存储器的层次结构

在同一台计算机中, 有各种工作速度、存储容量、访问方式、用途等均不相同的存储器, 这些存储器构成一个层次结构, 如图 3.6 所示。从上到下, 各种存储器的存储容量越来越大, 每位的价格越来越便宜, 但存储周期越来越长。

通用寄存器堆、指令和数据缓冲栈、一级 Cache 是在 CPU 芯片内部的, 它们的工作速度比较高。从二级 Cache(有时还有三级 Cache)以下, 是在 CPU 外部的, 工作速度逐级明

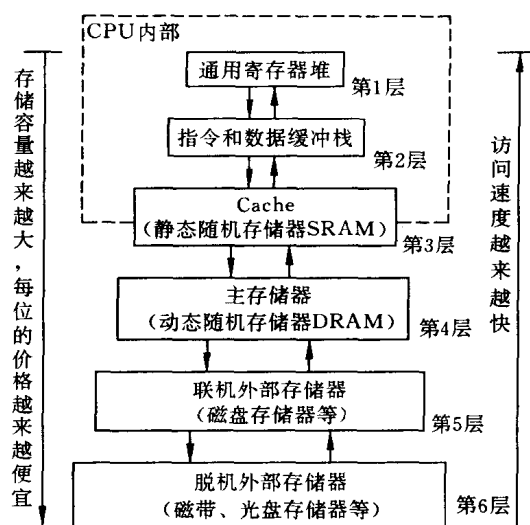


图 3.6 存储器的层次结构

显降低。如果用 i 表示层数，则有：

工作速度： $T_i < T_{i+1}$ ，用访问周期、存取周期或存取时间等表示。

存储容量： $S_i < S_{i+1}$ ，用字节 B、千字节 KB、兆字节 MB 和千兆字节 GB 等表示。

价格： $C_i > C_{i+1}$ ，用单位容量的价钱表示，例如，每一个二进制位多少美分（\$C/bit），每千字节多少美分（\$C/KB）等。

因为存放在各种存储器中的程序和数据最终都要送到 CPU 中进行处理，因此，从 CPU 的角度看，希望各层次存储器的工作速度是靠近上面的，存储容量和价格是靠近下面的。这正是存储系统要研究和得到的目标。

各级存储器的主要性能指标见表 3.1。这些指标仅供参考，因为随着时间的推移，有些指标变化非常快。

表 3.1 各种存储器的主要性能特性

存储器层次	通用寄存器	缓冲栈	Cache	主存储器	磁盘存储器	脱机存储器
存储周期	<10ns	<10ns	10ns~60ns	60ns~300ns	10ms~30ms	2min~20min
存储容量	<512B	<512B	8KB~2MB	32MB~1GB	1GB~1TB	5GB~10TB
价格(\$C/KB)	1 200	80	3.2	0.36	0.01	0.0001
访问方式	直接译码	先进先出	相联访问	随机访问	块访问	文件组
材料工艺	ECL	ECL	SRAM	DRAM	磁表面	磁、光等
分配管理	编译器分配	硬件调度	硬件调度	操作系统	系统/用户	系统/用户
带宽(MB/S)	400~8 000	400~1 200	200~800	80~160	10~100	0.2~0.6

1955 年，IBM 公司推出的第一台大型计算机 IBM704，CPU 和主存储器的工作周期均为 12 微秒，两者恰好能匹配工作。到了 40 多年后的今天，CPU 的工作速度提高了 4 个

数量级以上,而主存储器的工作速度仅提高两个数量级,两者根本不能匹配工作。因此,计算机系统结构的设计者们必须面对这一现实,找出解决的方法。以下将介绍解决这一问题的一些主要方法。

3.1.3 频带平衡

前面提到,现代计算机是以存储器为中心工作的。从图 3.1 中可以看到,一般计算机中的存储器有 4 个访问源。当然,这里所指的存储器可以广义地来理解,它不仅指主存储器(内存),也包括其他各种各样的存储器。由此可以看出,存储器的访问速度能不能跟上系统的需要,是影响整个计算机系统性能的极为重要的关键问题,这就是存储器的频带平衡问题。

使计算机系统中各级存储器的频带达到平衡,是系统结构设计者的一项重要工作。下面,先通过一个具体的例子来说明存储器的频带是如何计算的。

假设有一台速度为 200MIPS(每秒钟执行 2 亿条指令)的计算机,这在目前是很普通的机器。那么,各种访问源的频带计算如下:

CPU 取指令: 200MW/s (假设每条指令的长度为一个字(W))。

CPU 取操作数和保存运算结果: 400MW/s (平均每条指令访问两个操作数)。

各种输入输出设备访问存储器: 5MW/s 。

三项相加,要求存储器的频带宽度不低于 605MW/s 。如果存储器的字长就是一个字,则要求存储器的访问周期不大于 16.5ns 。但是,实际上目前作为主存储器的 DRAM(动态随机存储器)的工作周期为 200ns 左右(一般产品上标出的 60ns 是指数据读出时间,由于 DRAM 是一种破坏性读出的存储器,读出之后必须重新写入,另外,还有线路恢复和再生等都需要时间),两者相差 100 倍以上。

一般来说,有三条途径可以解决存储器的频带平衡问题。

1. 多个存储器并行工作,并且用并行访问和交叉访问等方法提高存储器的访问速度。具体方法将在下一节中介绍。

2. 设置各种缓冲存储器,例如,先行缓冲栈(lookahead),包括预取指令缓冲栈、操作数先行缓冲栈、运算结果后行缓冲栈等,有关原理将在第五章中介绍。就预取指令缓冲栈而言,如果能把一个短的循环程序都装入缓冲栈中,则在整个循环程序的执行过程中都不必到主存储器中取指令。

另外,通用寄存器实际上也是一种缓冲存储器,因为,存放在通用寄存器内的数据可以被重复使用。据统计,在许多程序中,数据在通用寄存器中被重复访问的次数平均达两次以上。这就大大缓解了对存储器的压力。

3. 采用存储系统,特别是 Cache 存储系统,这是目前计算机系统中提高存储器速度最有效的一种方法。一般计算机中都有两级 Cache 存储器,其中第一级在 CPU 芯片内部,速度很快,容量比较小,另一级在主板上,容量比较大,速度稍低些。

3.1.4 并行存储器

设置多个独立的存储器,让它们并行工作,在一个存储周期内可以访问到多个数据,

这是提高存储器速度最直接的方法。

以下,分别介绍并行访问存储器、交叉访问存储器和无访问冲突并行存储器三种并行存储器。

3.1.4.1 并行访问存储器

要在一个存储周期内访问到多个数据,最直接的办法是增加存储器的字长。例如,一般存储器在一个存储周期内只能访问到一个字。如图 3.7(a)所示,一个存储容量为 m 字 $\times w$ 位的存储器,每个存储周期只能访问到 w 位(一个字)。现在把存储器的字长增加 n 倍,成为 $n \times w$ 位,为了保持总的存储容量不变,可以把存储器的字数(也可以说是地址数)相应减少 n 倍,成为 m/n 个字。这样,在一个存储周期内就能访问到 n 个数据(每个数据的字长是 w 位),如图 3.7(b)所示。

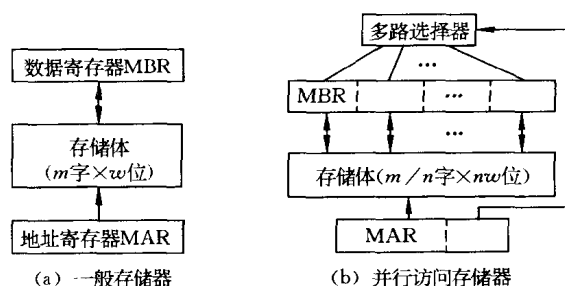


图 3.7 并行访问存储器与一般存储器比较

在具体实现时,要把地址码分成两个部分,其中一部分仍作为存储器的地址去访问存储器(因为存储器的字数减少了,因此访问存储器的地址码可以缩短),而另一部分则去控制一个多路选择器,从同时读出的 n 个数据中选择一个数据输出。

并行访问存储器的主要优点是实现非常简单、容易。主要缺点是访问的冲突大,主要冲突来自如下几个方面:

1. 取指令冲突。在遇到程序转移,而且转移成功时,一个存储周期中读出的 n 条指令中,后面的几条指令将无用。

2. 读操作数冲突。一次同时读出的 n 个操作数,不一定都有用。换一种说法,需要的多个操作数不一定正好都存放在同一个存储字中。

3. 写数据冲突。这种并行访问的存储器,必须凑齐了 n 个数之后才能一起写入存储器。如果只写一个字,必须先把属于同一个存储字的 n 数读入数据寄存器中,然后在地址码的控制下修改其中的一个字,最后再把整个存储字写回存储器。

4. 读写冲突。当要读出的一个字和要写入存储器的一个字处在同一个存储字内时,无法在一个存储周期内完成。

这 4 种冲突中,第 1 种冲突的概率比较小,因为,程序在大多数情况下是顺序执行的。第 2 种冲突的概率比较大,因为操作数的随机性比程序要大。第 3 和第 4 种冲突,解决起来有一定困难,需要专门进行控制。

分析发生这些冲突的原因,从存储器本身看,主要是因为地址寄存器和控制逻辑只有一套。如果在图 3.7(b)中有 n 个独立的地址寄存器和 n 套读写控制逻辑,那么,上述第 3 和第 4 种冲突就自然解决了,第 1 和第 2 种冲突也会有所缓解。

3.1.4.2 交叉访问存储器

交叉访问存储器通常有两种工作方式,一种是地址码高位交叉,另一种是地址码低位交叉。其中,只有低位交叉存储器才能有效地解决访问冲突问题。以下主要介绍低位交叉存储器,但由于高位交叉存储器目前使用得非常普遍,因此作简单介绍。

1. 高位交叉访问存储器

高位交叉访问存储器的工作原理如图 3.8 所示。地址码的低位部分是各个存储体的体内地址,高位部分用来区分存储体的体号。

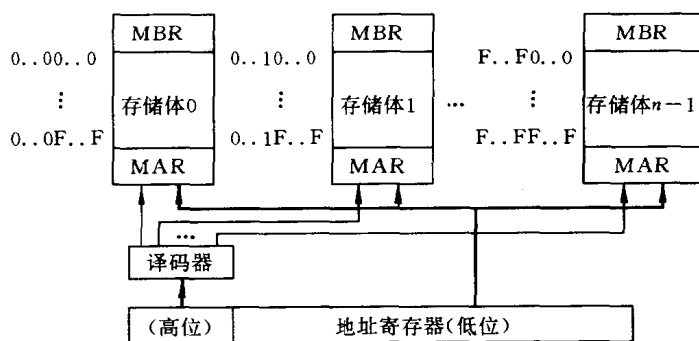


图 3.8 高位交叉访问存储器的结构

目前,大部分计算机系统的主存储器都采用模块化结构,用户可根据自己的不同需要随时很方便地改变主存储器的容量。例如,市场上可以买到 8MB、16MB、32MB、... 的主存储器模块。用两个 16MB 的模块可以构成一个 32MB 的主存储器,用 4 个 16MB 的模块也可以构成一个 64MB 的主存储器等。这种模块化的主存储器通常都是采用高位交叉编址方法构成的。

假设有一个采用高位交叉访问方式工作的主存储器,组成这个主存储器的各个存储体的参数如下:

m : 每个存储体的容量,通常用地址码的低 $\log_2 m$ 位作为存储体的体内地址,而且,每个存储体的体内地址均相同。

n : 组成存储器的存储器体个数的总和,通常用地址码的高 $\log_2 n$ 位作为一个译码器的输入,而这个译码器的 n 个输出作为各个存储体的使能输入端,只有被译码器选中的那个存储体才能进行读写操作。

j : 各个存储体的体内地址, $j=0,1,2,\dots,m-1$ 。

k : 组成主存储器的各个存储体的体号, $k=0,1,2,\dots,n-1$ 。

那么这个存储器的地址 A 的计算公式为: $A = m \times k + j$ 。

如果已知这个存储器的地址为 A ,可以计算出与这个地址对应的存储体的体号和它

下面举一个简单的例子来说明采用低位交叉访问方式工作的存储器,它的各个存储体的地址是如何编写的。一个由 8 个存储体构成的、总存储容量为 64 的主存储器的地址编写方法如图 3.10 所示。

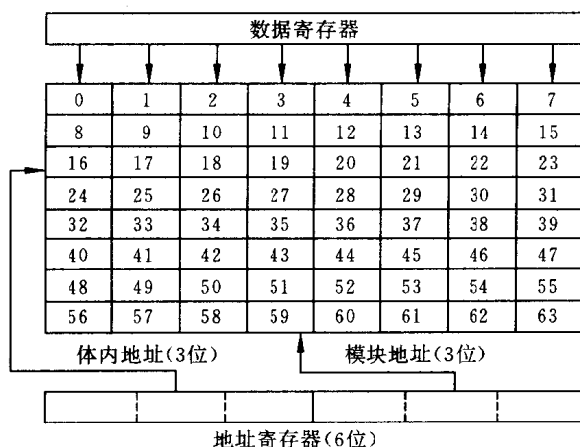


图 3.10 由 8 个存储体构成的主存储器的低位交叉编址方式

为了达到提高主存储器速度的目的,采用低位交叉访问方式的存储器,在一个存储器周期内, n 个存储体必须分时启动。启动的时间如图 3.11 所示。

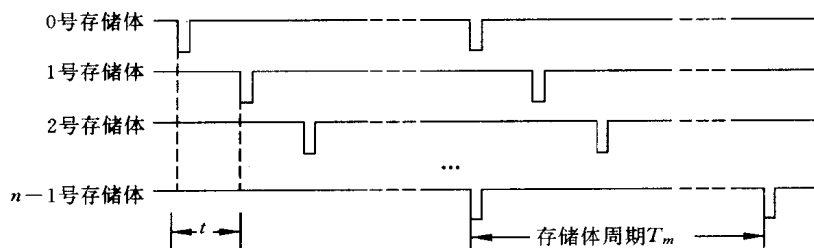


图 3.11 低位交叉编址主存储器的分时启动

如果每个存储体的访问周期为 T_m , 则由 n 个存储体构成的主存储器,各存储体的启动间隔 t 为:

$$t = \left\lfloor \frac{T_m}{n} \right\rfloor$$

从图 3.11 看到,采用低位交叉访问方式工作的存储器实际上是一种采用流水线方式工作的并行存储器系统。在连续工作的情况下,保持每个存储体的速度不变,而整个存储器的速度可望提高 n 倍。

目前的计算机系统一般都存在主存储器速度远远低于 CPU 速度的问题,在多处理机系统中这个问题尤为突出。除了采用多级高速 Cache 存储器之外,采用低位交叉访问方式的主存储器也是一种很好的方法。

采用低位交叉访问方式能够大幅度提高主存储器的速度,目前已经在共享主存储器

的多处理机系统中得到广泛应用,许多高速的单处理机也采用了低位交叉访问方式的存储器作为主存储器。

从直观上看,增加存储体个数(即增加 n 的值)能够提高主存储器的速度,但是,主存储器的速度不是随存储体个数的增加而线性提高的。例如,在有的大型计算机中采用 32 个存储体低位交叉来构成主存储器,而这个主存储器的速度只比单个存储体的速度高 10 倍左右。其根本原因是存在访问冲突问题,使得由 n 个存储体组成的主存储器的加速比通常要小于 n 。

下面,分析访问冲突的计算方法。

对于有 n 个存储体的主存系统,由于取指令时可能发生程序转移,取操作数时存在数据离散性,实际上每个存储周期只能取到 k 个有效字,其余 $n-k$ 个存储体存在地址冲突。 k 的最大值是 n ,最小值是 1, k 显然是一个随机变量。

假设 $p(k)$ 是 k 的概率密度函数,即 $p(1)$ 是 $k=1$ 的概率, $p(2)$ 是 $k=2$ 的概率, \dots , $p(n)$ 是 $k=n$ 的概率。 k 的平均值为:

$$N = \sum_{k=1}^n k \cdot p(k) \quad (3.4)$$

N 是 k 的平均值,实际上它就是在每个存储周期中能访问到的平均有效字的个数,以下,我们把它称为并行存储器的加速倍数(或加速比)。

概率 $p(k)$ 与具体程序的运行状况有密切关系。对于组成程序的指令序列,访问存储器的过程是按地址从小到大顺序进行的,因此,影响加速倍数 N 的主要原因是程序中的转移指令。为此,定义程序的转移概率为 g ,即从存储器中读出的指令是转移指令而且转移成功的概率。

$k=1$ 是指从第 1 个存储体中读出的指令就是转移指令,而且转移成功。根据上面的定义,这时有 $p(1)=g$ 。

$k=2$ 是指从第 1 个存储体中读出的不是转移指令,或者虽然是转移指令,但转移没有成功,其概率为 $1-p(1)$,即为 $1-g$ 。而从第 2 个存储体中读出的是转移指令,而且转移成功,因此, $p(2)=(1-p(1)) \cdot g$,即 $p(2)=(1-g) \cdot g$ 。

同理有:

$$p(3)=(1-p(1)-p(2)) \cdot g=(1-g)^2 \cdot g$$

$$p(k)=(1-g)^{k-1} \cdot g; \text{其中 } k=1,2,\dots,n-1$$

当 $k=n$ 时,表示从前 $n-1$ 个存储体中读出的都不是转移指令,或者转移都不成功,这时,不管从第 n 个存储体读出的是不是转移指令,加速倍数都是 N 。因此有:

$$p(n)=(1-g)^{n-1}$$

将这些概率表达式代入(3.4)关系式,得到:

$$N = 1 \cdot g + (1-g) \cdot g + 2 \cdot (1-g)^2 \cdot g + \dots + (k-1) \cdot (1-g)^{k-1} \cdot g \\ + (n-1) \cdot (1-g)^{n-1} \cdot g + n \cdot (1-g)^{n-1}$$

计算并化简后得:

$$N = \frac{1 - (1-g)^n}{g} \quad (3.5)$$

从(3.5)中可以看出,若 $g=0$,即连续 n 条指令都不是转移指令,或者,即使是转移指令,但转移没有成功,则 n 个存储体的加速倍数达到最高值 n 。若 $g=0$,即程序的每条指令都是转移指令,而且转移都成功,则 n 个存储体的加速倍数只有 1,与单个存储体完全一样。

表 3.2 给出 n 为 4、8、16 和 32, g 为 0.01、0.1、0.2、0.3 和 0.4 时,采用低位交叉访问方式的并行存储器的加速比分布情况。据统计,程序的转移概率 g 一般为 0.2 左右,从表中看,并行工作的存储体的个数 n 取不大于 8 为宜。当 n 大于 8 时,加速效果并不明显。

表 3.2 并行存储体个数与程序转移概率的关系

存储体个数	$g=0.01$	$g=0.1$	$g=0.2$	$g=0.3$	$g=0.4$
4	3.94	3.44	2.95	2.53	2.18
8	7.73	5.70	4.16	3.14	2.46
16	14.85	8.15	4.86	3.32	2.50
32	27.50	9.66	5.00	3.33	2.50

对于读操作数和写运算结果,随机性要比取指令大得多,因此,低位交叉访问存储器的加速倍数要比(3.5)关系式给出的还要低些。

然而,实际上不少机器取 n 大于 8,如 16 或 32 等。这时,只要有其他措施配合,加速倍数可以继续提高。例如,设置指令和数据缓冲存储器,并且把已经从存储器中读出的、暂时还没有用的指令或数据放入缓冲存储器,由于程序的局部性原理,只要缓冲存储器有一定的规模,这些指令和数据在以后还可能用得上。再例如,低位交叉访问方式与 Cache 配合,把一个存储周期中从主存储器中读出的 n 个字作为 Cache 的一块全部装入 Cache 中。主存储器与 Cache 每次交换数据都以这样的块为单位进行等。

另外,前面介绍的并行访问存储器,当设置了缓冲存储器或 Cache 之后,访问冲突的概率也会明显降低。

表 3.3 给出早期的几台大型计算机的主存储器结构,包括低位交叉访问的存储体个数、并行访问的每个存储体字长、存储器频带宽度等。

表 3.3 几台大型计算机的主存储器结构

机器型号	存储体个数 n	存储体字长 w	存储周期 T_m	频带宽度 B_m
IBM370/165	4	32	2 000ns	8MB/s
CDC6600	32	32	1 000ns	128MB/s
CDC7600	32	32	275ns	465MB/s
CRAY-1	16	64	50ns	2 560MB/s
ASC	8	256	160ns	1 600MB/s
STAR-100	32	512	1 280ns	1 600MB/s

从以上分析中看出,采用并行访问存储器和交叉访问存储器虽然能提高主存储器的

速度,但是当存储体个数 n 增加到一定数量时,加速效果并不明显。因此,必须把并行和交叉访问存储器与 Cache 存储系统等配合起来才能组成高速度的主存储器。

3.1.4.3 一种无访问冲突存储器

采用低位交叉访问技术时,一个由 n 个存储体构成的主存储器,它的速度并不能提高 n 倍,其根本原因是存在访问冲突。产生访问冲突的根源主要有两个,一是程序中有转移指令,二是数据的随机性。从上面的分析结果看,后一个问题更为严重。

下面,介绍多维数组的无冲突访问存储器。

先举一个一维数组的例子。如果采用低位交叉访问方式的并行存储器有4个存储体,交叉存放一维数组 a_0, a_1, a_2, \dots ,如图3.12所示。

	0号体	1号体	2号体	3号体
体内地址 0	a_0	a_1	a_2	a_3
1	a_4	a_5	a_6	a_7
2	a_8	a_9	a_{10}	a_{11}
3

图 3.12 一维数组的存储方案

如果每次都按连续地址访问,就没有冲突问题。一个存储周期可以读出4个操作数,若一个存储周期需要读出更多的操作数,只要增加存储体的个数就可以了。如果要求按位移量为2的变址方式访问并行存储器(读下标为奇数或偶数的操作数),则存储器的频带宽度就要降低一半,即可能有一半地址是冲突的。在并行递归算法中,典型的访问模式是向量子集的各个元素逐次按2的整数幂相间访问。例如,先按地址连续访问,然后按位移量为2的变址方式访问,再按位移量为4的变址方式访问等。对于这类算法,一般的低位交叉访问存储器就显得不能适应了。

只要认真分析造成访问冲突的原因,不难发现传统的低位交叉访问存储器的存储体个数 n 为2的整数幂,因此变址位移量正好是 n 的约数。解决这一问题的方法很简单,只要把存储体的个数 n 选为质数,变址位移量就必然与 n 互质,一维数组的访问冲突自然也就不存在了。

许多以向量计算为主要任务的大型计算机系统,其主存储器的存储体个数一般都是质数。例如,美国Burroughs公司研制的巨型科学处理机BSP,它的存储体个数为17,我国研制的银河巨型计算机,存储体的个数为31。

对于多维数组,需要考虑的问题就复杂得多,为简单起见,下面仅讨论二维数组。讨论过程中所得出的结论也可以推广到多维数组中。

假设一个 $n \times n$ 的二维数组存储在一个并行存储器中。现在要求对这个二维数组实现按行、按列、按对角线和按反对角线访问,并且在不同的变址位移量情况下,都能实现无冲突访问。

下面以 4×4 的二维数组为例来讨论这个问题。如果按照地址顺序存储,一个 4×4 二维数组存储在4个并行存储体中的情况如图3.13(a)所示。显而易见,对于按行、按对角

线访问能做到不发生冲突。但是,若按列访问,由于4个元素都放在同一个存储体内,只能分4个存储周期顺序读取它们。

为了实现按行和按列都能无冲突访问,有一种存储方案是将数组的各个元素在4个并行存储体中错位存放,如图3.13(b)所示。但是,这种存储方案造成按对角线访问时又发生冲突了。

	0号体	1号体	2号体	3号体
体内地址 0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}
3	a_{30}	a_{31}	a_{32}	a_{33}

(a) 按列访问有冲突的存储方案

	0号体	1号体	2号体	3号体
体内地址 0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{13}	a_{10}	a_{11}	a_{12}
2	a_{22}	a_{23}	a_{20}	a_{21}
3	a_{31}	a_{32}	a_{33}	a_{30}

(b) 按对角线访问有冲突的存储方案

图 3.13 4×4 数组有访问冲突的存储方案

参照一维数组的情况,不难证明,如果二维数组的各个元素采用按地址顺序存储的方案,当并行存储体的个数是偶数时,对于 $n \times n$ 二维数组就不可能实现按行、按列、按对角线和反对角线的无冲突访问。当然,也可以针对不同的数组采取各种变通的措施,例如,采用不同的错位存储方案等。但是,这会给编译程序增加很大的麻烦,硬件实现的代价也很高。因此,希望寻找到一种对用户来说最为方便的均匀无冲突存储方案,能够实现二维数组的按行、按列、按对角线和反对角线的变址无冲突访问,而且,硬件实现比较简单,代价不要太高。

对于 $n \times n$ 的二维数组,P. Budnik 和 D. J. Kuck 提出了一种能够实现上述要求的无冲突存储方案。这种方案要求并行存储体的个数 $m \geq n$,并且取质数,同时还要在行、列方向上错开一定的距离存储数组元素。设同一列相邻元素在并行存储器中错开 d_1 个存储体存放,同一行相邻元素在并行存储器中错开 d_2 个存储体存放。当 $m = 2^{2^p} + 1$ (p 为任意自然数)时,能够同时实现按行、按列、按对角线和按反对角线无冲突访问的充要条件是: $d_1 = 2^p, d_2 = 1$ 。

以 4×4 的二维数组为例,取大于4的最小质数 $m = 5$ 作为并行存储体的个数,并把 m 代入关系式 $m = 2^{2^p} + 1$,解得到 $p = 1$,从而计算得到 $d_1 = 2^1 = 2, d_2 = 1$ 。一个 4×4 的二维数组在5个存储体中的存储情况如图3.14所示。原来在同一列中的相邻元素要错开两个存储体存放,如 a_{10} 要存放在2号存储体中, a_{20} 要存放在4号存储体中,……。原来在同一行中的各个元素仍然按顺序存放在该行中,但要按5取模。

	0 号体	1 号体	2 号体	3 号体	4 号体
体内地址 0	a_{00}	a_{01}	a_{02}	a_{03}	
1	a_{13}		a_{10}	a_{11}	a_{12}
2	a_{21}	a_{22}	a_{23}		a_{20}
3		a_{30}	a_{31}	a_{32}	a_{33}

图 3.14 4×4 二维数组按行、列、对角线和反对角线访问都不冲突的存储方案

很明显,按照图 3.14 所示的存储方案, 4×4 二维数组在按行、列、对角线和反对角线访问时均没有冲突。并且,由于并行存储体的个数是质数,对于不同的变址位移量,对这个数组的访问仍然是没有冲突的。

假设 a_{ij} 是 $n \times n$ 二维数组中的任意一个元素,则这个元素在无冲突并行存储器中的体号地址和体内地址可以通过如下的一般公式来计算:

$$\text{体号地址: } (2^p i + j + k) \text{ MOD } m$$

$$\text{体内地址: } i$$

其中, $0 \leq i \leq n-1, 0 \leq j \leq n-1, k$ 是数组的第一个元素 a_{00} 所在体号地址,一般情况下取 $k=0$; m 是并行存储体的个数,要求 $m \geq n$ 且为质数; p 是满足 $m = 2^{2^p} + 1$ 关系的任意自然数; MOD 是模运算的符号。

在上述 $n \times n$ 二维数组的并行存储方案中,有 $1/(n+1)$ 的存储单元是浪费的,这种存储方案实际上浪费了一个存储体。如果不要求同一行中的数组元素按地址顺序存储,则 $n \times n$ 的二维数组实际上只需要用 n 个并行存储体就能实现按行、列、对角线和反对角线的无冲突访问。这时,并行存储器的利用率最高,没有浪费的存储单元。

对于任意一个 $n \times n$ 的二维数组,如果能够找到满足 $n = 2^{2^p}$ 关系的任意自然数 p ,则这个二维数组就能够使用 n 个并行存储体实现按行、列、对角线和反对角线的无冲突访问。下面仍以 4×4 的二维数组为例,介绍如何计算数组元素的体号地址和体内地址。

假设 a_{ij} 是 4×4 二维数组中的任意一个元素,其中的下标 i 和 j 都可以用两位二进制数表示,假设 i 和 j 的高位和低位分别为 i_H, i_L, j_H 和 j_L ,则 a_{ij} 在无冲突并行存储器中的体号地址和体内地址可以通过如下公式计算:

$$\text{体号地址: } 2(i_L \oplus j_H) + (i_H \oplus i_L \oplus j_L)$$

$$\text{体内地址: } j$$

其中, $0 \leq i \leq 3, 0 \leq j \leq 3, i_H, i_L, j_H, j_L$ 均为 0 或 1。

根据这个体号地址和体内地址计算公式,一个 4×4 二维数组在 4 个并行存储体中的存储方案如图 3.15 所示。很容易验证这种存储方案能够实现按行、列、对角线和反对角线的无冲突访问。通过卡诺图很容易证明上述体号地址和体内地址计算公式是正确的。

当数组的维数 n 大于 4 时,可以把数组划分成多个 4×4 的子阵。只要每个子阵都能实现按行、列、对角线和反对角线的无冲突访问,则整个数组必然也能实现这种无冲突访问。例如,可以把一个 16×16 二维数组划分成 4 个 4×4 子阵。实际上,在图 3.15 中的每

个 2×2 子阵也都能实现按行、列、对角线和反对角线的无冲突访问。

	0 号体	1 号体	2 号体	3 号体
体内地址 0	a_{00}	a_{20}	a_{30}	a_{10}
1	a_{21}	a_{01}	a_{11}	a_{31}
2	a_{32}	a_{12}	a_{02}	a_{22}
3	a_{13}	a_{33}	a_{23}	a_{03}

图 3.15 4×4 数组用 4 个存储体的无访问冲突存储方案

比较图 3.14 和图 3.15 两种二维数组的并行存储方案,两者的效果相同,都能实现按行、列、对角线和反对角线的无冲突访问。前一种方案多了一个存储体,在整个并行存储器中有 $1/5$ 的存储单元是浪费的;然而,这种存储方案也有一个明显的优点,在读或写并行存储器时所需要的对准网络非常简单。后一种方案虽然少了一个存储体,没有浪费的存储单元,但是在执行并行读和并行写操作时需要借助比较复杂的对准网络。例如,在读并行存储器时,要通过一个对准网络把数组元素的地址变换成并行存储器的实际地址,读出的数据要通过另一个对准网络恢复成原来的顺序。因此,在实际应用时要根据具体情况决定采用哪一种存储方案。另外,在一般通用计算机的并行存储器中还存储有大量的一维数组和程序等,如前面所述,只要并行存储体的个数为质数,一维数组能够很容易实现不同位移量的变址无冲突访问,而且不会有存储单元的浪费现象。

3.2 虚拟存储器

虚拟存储器又称虚拟存储系统,或虚拟存储体系等,其概念是 1961 年由英国曼彻斯特大学的 Kilbrn 等人提出的,到了 70 年代被广泛地应用于大中型计算机系统中。目前,许多小型计算机,甚至微型机也开始使用虚拟存储器。

虚拟存储器由主存储器和联机工作的外部存储器共同组成。在目前的计算机系统中,主存储器通常用动态随机存储器(DRAM)实现,它的存储容量相对比较小,速度比较快,单位容量的价格比较贵。联机工作的外部存储器通常为磁盘存储器,它的存储容量很大,与主存储器相比,速度很低,单位容量的价格很便宜。这两个存储器在硬件和系统软件的共同管理下,对于应用程序员,可以把它们看作是一个单一的存储器,是一个存储容量非常大的主存储器。

以下首先介绍虚拟存储器的基本工作原理,然后介绍虚拟存储器的地址映象和变换方法,页面替换算法,加快地址变换速度的方法等,最后分析影响虚拟存储器性能的几个主要问题。本书主要从计算机系统结构的角度来讲述虚拟存储器,这与《计算机操作系统》课中从“存储器管理”这一角度来介绍虚拟存储器是很不一样的。可以说,只有当你学习了《计算机系统结构》之后,才能真正懂得虚拟存储器的原理和实现方法。

3.2.1 虚拟存储器工作原理

页式虚拟存储器是虚拟存储器中用得比较广泛的一种,另外的段式虚拟存储器和段

页式虚拟存储器主要是因为地址变换方法不同产生的。本节首先以页式虚拟存储器为例介绍虚拟存储器的工作原理。然后再具体介绍页式、段式和段页式三种虚拟存储器的地址变换方式及外部地址变换方式。

就像一本书由很多页组成一样,在页式虚拟存储器中,把主存储器、磁盘存储器和虚拟存储器都划分成固定大小的块——页(page),每一页中容纳的字数是相同的。主存储器的页称为实页,虚拟存储器中的页称为虚页。一个主存地址 A 由两部分组成,实页号 p 和页内偏移 d,如图 3.16(a)所示。一个虚地址 A_v 由三部分组成,用户号 U、虚页号 P 和页内偏移 D,如图 3.16(b)所示。

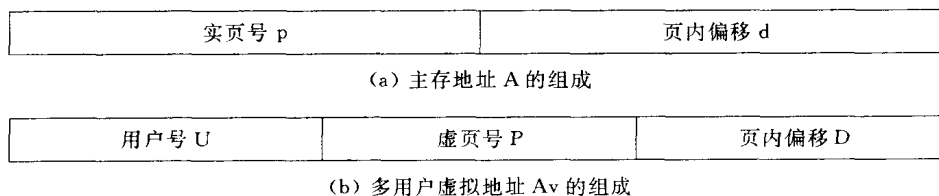


图 3.16 虚拟存储器中的地址组成

一个用户程序要访问虚拟存储器时,必须给出多用户虚拟地址 A_v。在操作系统和有关硬件的共同管理下,首先进行内部地址变换。如果变换成功(命中),得到主存实页号 p。而多用户虚拟地址中的页内偏移 D 可以直接作为主存实地址中的页内偏移 d,这样,只要把主存实页号 p 与它的页内偏移 d 直接拼接起来就得到主存实地址 A。于是,就可以用这个主存实地址 A 去访问主存储器,得到所需要的数据。页式虚拟存储器的工作过程如图 3.17 所示。

如果内部地址变换失败(未命中),表示要访问的数据不在主存储器中,必须访问磁盘存储器。这时,要进行外部地址变换。外部地址变换主要用软件实现。首先查外页表得到与虚页号 P 相对应的磁盘存储器的实地址,然后再查内页表(主存实页表),看主存储器中是否有空页。如果主存储器中还有空页,只要找到空页号。把磁盘存储器的实地址和主存储器的实页号送入输入输出处理机(输入输出通道)等,在输入输出处理机的控制下,把要访问数据所在的一整页都从磁盘存储器调入到主存储器。

如果主存储器中已经没有空页,则要采用某种页面替换算法,先把主存中暂时不用的一页写回到磁盘存储器中原来的位置上,以便腾出空位置来存放新的页。当然,如果要替换出去的那一页自从被调入主存储器之后从来没有被修改过,就不需要把它送回磁盘存储器,直接用调入的新页把它覆盖掉即可。

在进行外部地址变换时,如果没有命中,则表示所需要的页不在磁盘存储器中。这时就要在操作系统控制下,启动磁带机、光盘存储器等海量存储器,把与所需要数据相关的文件调入磁盘存储器。

3.2.2 地址的映象与变换

在虚拟存储器中有三种地址空间,一种是虚拟地址空间,也称虚存空间或虚拟存储器空间,它是应用程序员用来编写程序的地址空间,这个地址空间非常大。第二种是主存储

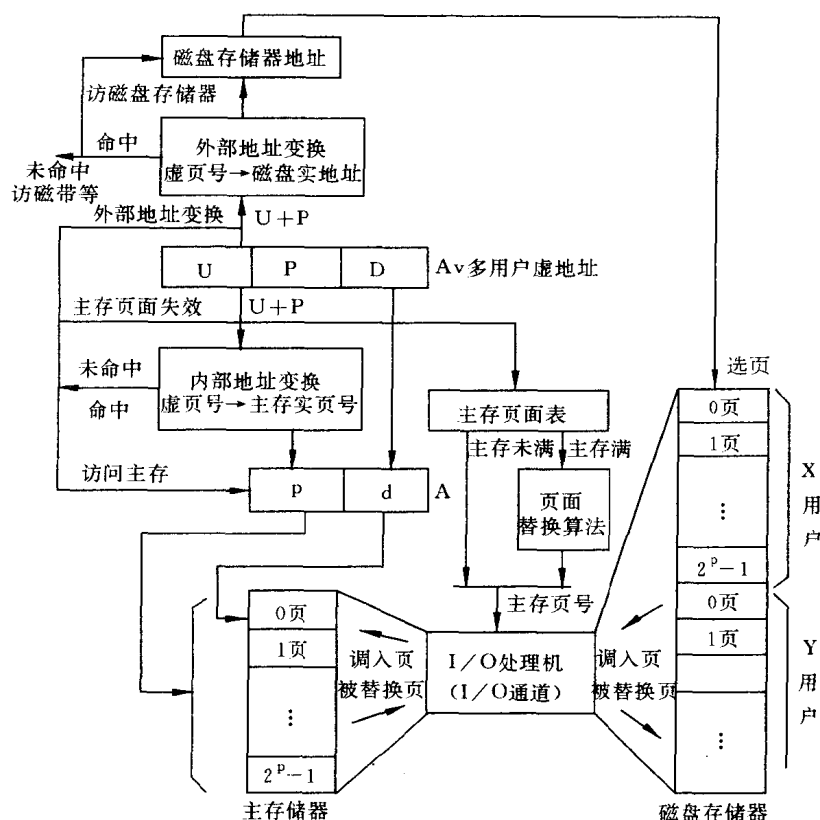


图 3.17 页式虚拟存储器工作原理

器的地址空间,也称主存地址空间、主存物理空间或实存地址空间。第三种是辅存地址空间,也就是磁盘存储器的地址空间。与这三种地址空间相对应,有三种地址,即虚拟地址(虚存地址、虚地址)、主存地址(主存实地址、主存物理地址、主存储器地址)和磁盘存储器地址(磁盘地址、辅存地址)。

地址映象是把虚拟地址空间映象到主存地址空间,具体地说,就是把用户用虚拟地址编写的程序按照某种规则装入到主存储器中,并建立多用户虚地址与主存实地址之间的对应关系。而地址变换则是在程序被装入主存储器之后,在实际运行时,把多用户虚地址变换成主存实地址(内部地址变换)或磁盘存储器地址(外部地址变换)。

根据所采用的地址映象和地址变换方法不同,有多种不同类型的虚拟存储器。目前主要有页式虚拟存储器、段式虚拟存储器和段页式虚拟存储器等三种。然而,这三种虚拟存储器的基本原理、所采用的主要技术和工作流程等是基本相同的。

3.2.2.1 段式虚拟存储器

程序员在编写程序时,一般按照程序的内容和函数关系把程序分成段,每段都有自己的名字,并且希望能够按照名称或序号来访问程序段。采用段式虚拟存储器能够比较好地

满足程序员的这些要求。

程序段可以是主程序,也可以是各种子程序,还可以是数据块、数组、表格、向量等。每个程序段都从 0 地址开始编址,其长度可长可短,甚至可以在程序执行时动态地改变程序段的长度。

每一道程序(或一个用户、一个进程等)由一张段表控制,每个程序段在段表中占一行。段表的内容主要包括段号(或段名)、该程序段的长度、在主存中的起始地址等三个字段。其中,如果第一个字段用段号而不用段名表示,而且段号是连续的,则这一个字段可以省掉。只要根据段表中的另外两个字段:该程序段在主存中的起始地址和段的长度,就可以把该程序段唯一地映象到主存储器的确定位置中。另外,根据需要还可以在段表中增加其他信息,如指出该程序段的访问方式(可读可写、只读、某些用户可写、只能执行等)、是否已经装入主存的标志、该段程序是否被修改过的标志等,不过,增加的这些字段在地址映象过程中是用不上的,可以用于地址变换过程中。

图 3.18 是段式虚拟存储器的地址映象方法。一个由 4 个程序段组成的程序,在一张段表的控制下,把这 4 段程序分别映象到主存储器的各个不同区域中。每一个程序段可以映象到主存储器的任意位置,可以连续存放,也可以不连续存放,可以顺序存放,也可以前后倒置等。

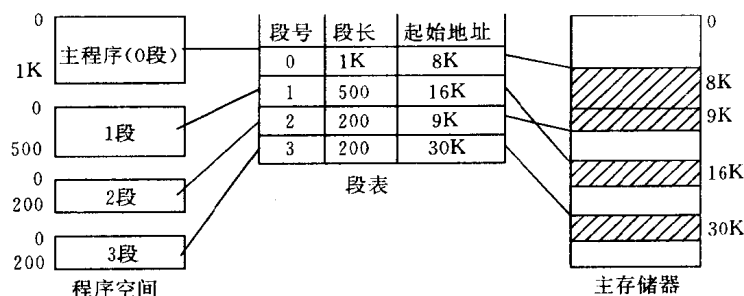


图 3.18 段式虚拟存储器的地址映象

使用地址映象方法,可以把在虚拟地址空间中编写的程序装入到主存储器中。但在程序实际执行时,还要把用来访问主存储器的多用户虚地址变换成主存实地址,有了主存实地址才能访问已经装在主存储器中的用户程序或数据,地址变换过程如图 3.19 所示。一个多用户虚地址由三部分组成,用户号 U(或程序号)、段号 S 和段内偏移 D。在 CPU 内通常有一个段表基址寄存器堆,每道程序使用其中的一个基址寄存器。因此,可以通过多用户虚地址中的用户号直接找到与这道程序相对应的基址寄存器。通常,段表是放在主存储器中的,从这个基址寄存器中可以直接读出段表的起始地址,把这个起始地址与多用户虚地址中的段号相加就能得到这个程序段的段表地址。访问这个段表地址,就能得到有关该程序段的全部信息。如果装入位给出的信息表示要访问的这个程序段已经在主存储器中,这时,只要把段表中给出的该程序段的起始地址与多用户虚地址中的段内偏移 D 相加就能得到主存实地址。

段表中的段长和访问方式是用来保护程序段的。可以根据程序段的起始地址和段长

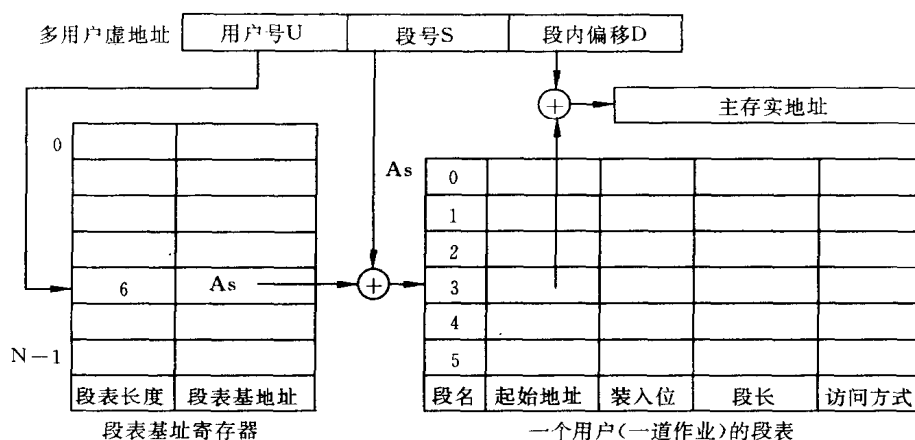


图 3.19 段式虚拟存储器的地址变换

计算出本次访问主存储器的地址是否越界。访问方式可以指出本程序段是否需要保护和保护的级别。例如,对于子程序段,通常只能执行,不能改写;对于一些常数段或数据库中的数据段,一般用户程序只能读,不能改写,不能执行;对于有些需要保密的表格,一般用户应禁止访问等。

如果装入位给出的信息表示要访问的这个程序段不在主存储器中,则段表中的起始地址和访问方式字段等均无用。这时,可以把它们用来存放该程序段在磁盘存储器中的起始地址等信息。使用磁盘存储器的起始地址和段长就可以从磁盘存储器中把该程序段读到主存储器中来。

根据需要还可以在段表中增加其他字段,例如,增加一个修改标志字段,表示本程序段是否被修改过。如果这个程序段从装入主存储器起一直没有被修改过,则在需要把它替换出主存储器时,不必把它写回到外部的磁盘存储器中,只要用新调入的程序段把它覆盖掉即可。如果这个程序段被修改过,则必须先把这个程序段全部写回到磁盘存储器中存放这个程序段的原来位置上。

段表本身也是一个段,一般常驻在主存储器中。如果段表太长,也可以把暂时不用的一部分段表放在磁盘存储器中,当需要时再把有用的段表调入主存储器。

段式虚拟存储器的主要优点如下:

1. 程序的模块化性能好。对于大程序,可以划分成多个程序段,每个程序段赋予不同的名字,由多个程序员并行编写,分别编译和调试。由于各个程序段在功能上是相互独立的,因此,一个程序段的修改和增删等不会影响其他程序段,从而可以缩短程序的编制和调试时间。

2. 便于程序和数据的共享。由于程序段是按功能来划分的,如子程序段、数组段、表格段等。每个程序段有比较完整的功能,因此,被共享的可能性很大。当某个程序段需要被共享时,只要在主存储器中装入一份,同时在需要调用这个程序段的那些程序(或用户)的段表中都使用这个程序段的主存起始地址和段长等信息,就能很方便地实现程序段的共享。

3. 程序的动态链接和调度比较容易。由于每一个程序段都是一组有独立意义的数据块或具有完整功能的程序段,因此,在程序运行过程中,可以根据需要一次就把一个程序段或数据块都装入到主存储器中,并且在装入时才实行动态链接。

4. 便于实现信息保护。在一般情况下,一段程序是否需要保护是根据这个段程序的功能来决定的。例如,一般的数据段能读能写,常数段只能读不能写,所有的数据段都不允许执行,而子程序一般只能执行不能修改等。由于段式虚拟存储器本身就是按照功能划分程序段的,因此,只要在段表中设置一个信息保护字段,就能根据需要很方便地实现对该程序段的保护。

段式虚拟存储器的主要缺点是:

1. 地址变换所花费的时间比较长。从图 3.19 中看到,从多用户虚地址变换到主存实地址需要查两次表,做两次加法运算。

2. 主存储器的利用率往往比较低。由于每个程序段的长度是不同的,一个程序段通常要装在一个连续的主存空间中,程序段在主存储器中不断地调入调出,有些程序段在执行过程中还要动态增加长度,从而使得主存储器中有很多的空隙存在。当然,也可以采用一些好的算法来减少空隙的数量,或者通过定时运行回收程序来合并这些空隙,但这无疑增加了系统的开销。

3. 对辅存(磁盘存储器)的管理比较困难。磁盘存储器通常是按固定大小的块来访问的,如何把不定长度的程序段映象到固定长度的磁盘存储器中,需要做一次地址变换。

3.2.2.2 页式虚拟存储器

在段式虚拟存储器中,虚拟地址空间按段划分,段的长度根据程序的逻辑需要可以是任意长的,主存储器的地址空间仍保持一维连续线性空间不划分。因此,段表中主存地址和段长度两个字段的长度都很长。这样,既增加了硬件开销,又降低了地址映象和变换的速度。

页式虚拟存储器把虚拟地址空间划分成一个个固定大小的块,每块称为一页(page),把主存储器的地址空间也按虚拟地址空间同样的大小划分为页。页是一种逻辑上的划分,它可以由系统管理软件任意指定。然而,由于磁盘存储器的物理块大小是 0.5KB,为了与外部存储器、特别是磁盘存储器相配合,虚拟存储器中页的大小通常指定为 0.5KB 的整数倍。目前在一般计算机系统中,一页的大小通常为 1KB 至 16KB。

在虚拟存储器中,虚拟地址空间中的页称为虚页,主存地址空间的页称为实页。这样,把一个在虚拟地址空间中编写的用户程序映象到主存实地址空间时,只需要建立从虚页号到实页号的地址变换即可,从而可以大大缩短地址的长度,既节省了硬件,又能加快地址变换的速度。

页式虚拟存储器的地址映象方法如图 3.20 所示。一个在虚拟地址空间中编写的用户程序共有 3 页,其中最后一页没有满,要浪费掉一部分。页表共有 4 行,每行与用户程序的一页对应。如果页表中第一个字段给出的页号是连续的,则这一个字段可以省掉。只要根据页表中的主存页号就能把该用户程序的每一页唯一地映象到主存储器的确定位置中。用户程序的每一页都可以映象到主存储器的任意一页的位置上,页与页之间可以连续映

象,也可以间断映象,可以按顺序来映象,也可以前后倒置等。

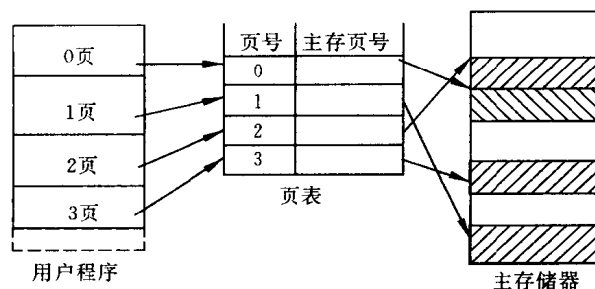


图 3.20 页式虚拟存储器的地址映象

与段式虚拟存储器相比,由于每一页的长度是固定的,因此,不需要像段式虚拟存储器中的段长度这一字段。另外,主存地址这一字段只需要指出主存储器的页号,与段式虚拟存储器中的主存地址必须指出整个主存地址长度相比要节省很多。

地址映象把用户用虚拟地址编写的程序装入了主存实地址空间中,但在程序运行过程中,还必须把用户程序中的虚拟地址变换成主存实地址,这是地址变换要做的工作。页式虚拟存储器的地址变换过程如图 3.21 所示。一个多用户虚地址由三部分组成:用户号 U、虚页号 P 和页内偏移 D。

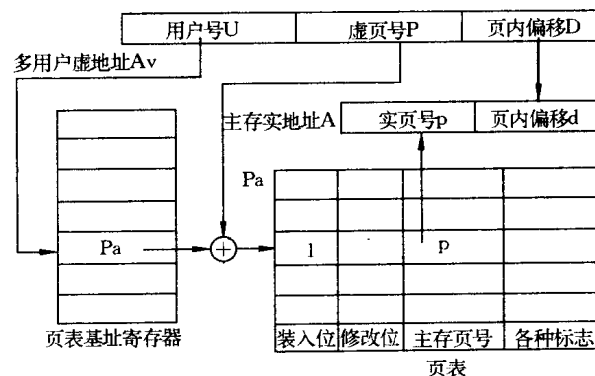


图 3.21 页式虚拟存储器的地址变换

在 CPU 内部有一个基址寄存器堆,用来存放页表的基地址,每个用户(每道程序)使用其中的一个基址寄存器。通过多用户虚地址中的用户号 U 可以直接找到与这个用户程序相对应的基址寄存器,从这个基址寄存器中读出页表起始地址。这个用户程序的每一页在页表中都有对应的一行。由于页表是按照用户程序的页号顺序存放的,因此,可以省掉页号这一字段。要找到被访问页的页表地址,只要把页表起始地址与多用户虚地址中虚页号相加即可。访问这个页表地址,就能得到被访问页的所有信息。把得到的主存页号 p 与多用户虚地址中的页内偏移 D 直接拼接起来得到主存实地址 A。

页表中的其他信息,如装入位、修改位和各种标志信息等的含义和用法与段式虚拟存

储器基本相同。

页式虚拟存储器的主要优点是：

1. 主存储器的利用率比较高。每个用户程序只有不到一页(平均为半页)的浪费,与段式虚拟存储器每两个程序段之间都有浪费相比要节省许多。

2. 页表相对比较简单。它需要保存的字段数比较少,一些关键字段的长度要短许多,因此,节省了页表的存储容量。

3. 地址映象和变换的速度比较快。在把用户程序装入到主存储器的过程中,只要建立用户程序的虚页号与主存储器的实页号之间的对应关系即可,不必使用整个主存的地址长度,也不必考虑每页的长度等。在地址变换过程中,从图 3. 21 中可以看到,主存实地址 A 是由页表中的主存页号 p 和多用户虚地址中的页内偏移 D 直接拼接而得到的,不必经过任何运算,因此,地址变换的速度比较快。

4. 对辅存(磁盘存储器)的管理比较容易。因为页的大小一般取磁盘存储器物理块大小(512 字节)的整倍数。

页式虚拟存储器的缺点主要有两个：

1. 程序的模块化性能不好。由于用户程序是强制按照固定大小的页来划分的,而程序段的实际长度一般是不固定的。因此,页式虚拟存储器中一页通常不能表示一个完整的程序功能。一页可能只是一个程序段中的一部分,也可能在一页中包含了两个或两个以上程序段。

2. 页表很长,需要占用很大的存储空间。通常,虚拟存储器中的每一页在页表中都要占有一个存储字。假设有一个页式虚拟存储器,它的虚拟存储空间大小为 4GB,每一页的大小为 1KB,则页表的容量为 4M 存储字。如果每个页表存储字占用 4 个字节,则页表的存储容量为 16MB。

3. 2. 2. 3 段页式虚拟存储器

为了能够同时获得段式虚拟存储器在程序模块化方面的优点和页式虚拟存储器在管理主存和辅存物理空间方面的优点,把两种虚拟存储器结合起来就成为段页式虚拟存储器。其基本思想是对用户用来编写程序的虚拟存储空间采用分段的方法管理,而对主存储器的物理空间采用分页的方法管理。

段页式虚拟存储器一方面具有段式虚拟存储器的主要优点,例如,用户程序可以模块化编写,程序段的共享和信息的保护都比较方便,程序可以在执行时再动态链接等。另一方面也具有页式虚拟存储器的主要优点,例如,主存储器的利用率比较高,对辅助存储器(磁盘存储器)的管理比较容易等。

在段页式虚拟存储器中,用户仍然按照逻辑的程序段来编写程序,但每一个程序段又被分成若干个固定大小的页。如图 3. 22 所示,一个用户程序由三个独立的程序段组成。0 号程序段的长度为 12KB,由于页的长度是 4KB,因此,正好分成 3 页。1 号程序段的长度为 10KB,也分成 3 页,其中最后一页有 2KB 是浪费的。2 号程序段的长度为 5KB,分成 2 页,其中后面一页浪费 3KB。

一个用户程序的三个程序段通过一张段表来控制。与段式虚拟存储器一样,每个程序

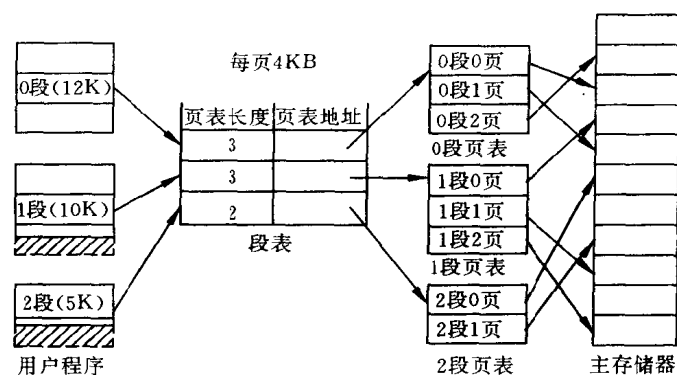


图 3.22 段页式虚拟存储器的地址映射

段在段表中占一行。在段表中给出该程序段的页表长度和页表的起始地址，根据这两个参数就能找到这个程序段的页表。页表的长度就是这个程序段的页数，页表中给出这个程序段的每一页在主存储器中的实页号。这样就完成了用户程序的虚拟空间到主存实地址空间的映射。

在虚拟存储空间到主存物理空间的映射过程中，是以页为单位进行的。与段式虚拟存储器不同的地方是：用户程序中每一页不能被映射到主存储器的任意开始位置上，只能映射到主存储器的一个整页的位置中。这种映射方法可以缩短页表的存储容量，加快地址映射和变换的速度，因为主存实地址只需要把页表中的实页号和虚拟地址中的页内偏移拼接起来即可，不必进行任何运算。

在段页式虚拟存储器中，一个多用户虚地址由四部分组成：用户号 U、段号 S、虚页号 P 和页内偏移 D，如图 3.23 所示。在程序运行过程中，要把用户程序中的虚拟地址变换成

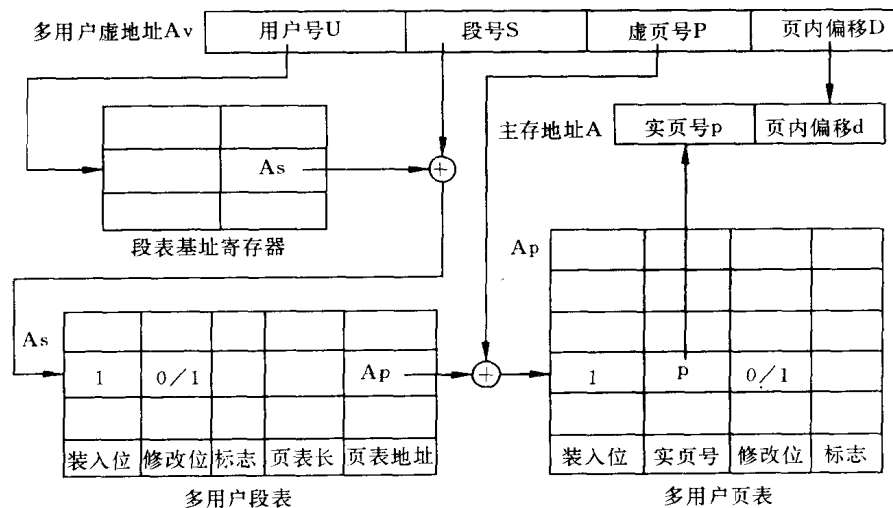


图 3.23 段页式虚拟存储器的地址变换

主存实地址,必须分两步进行。先查段表,得到该程序段的页表起始地址和页表长度,再通过查页表找到要访问的主存实页号,最后把实页号 p 与页内偏移 d 拼接得到主存的实地址。地址变换的具体过程可以参照段式虚拟存储器和页式虚拟存储器,以及图 3.23 自行分析清楚,这里不再详细叙述。

许多大中型计算机,如 IBM370/168、Multics、Amdahl470V/6 等都采用这种段页式虚拟存储器。

由图 3.23 可以看到,在段页式虚拟存储器中,要从主存储器中访问一个数据(取指令、读操作数或写结果),需要查两次表,一次是页表,另一次是段表。如果段表和页表都是在主存储器中的,则要访问主存储器三次。对于段式虚拟存储器和页式虚拟存储器也要访问主存储器两次。因此,要想使虚拟存储器的速度接近主存储器的速度,或者说,要想使虚拟存储器能够真正实用,必须加快查表的速度。有关具体方法,将在下一节中作比较详细的介绍。

3.2.2.4 外部地址变换

以上介绍了内部地址映象和变换方法,即把虚拟地址空间映象到主存物理地址空间、以及把虚拟地址变换成主存实地址的方法。当页表或段表中的有效位指示发生页面失效时,表示需要访问的那一页或那一个程序段还没有装入到主存储器中,这时必须进行外部地址变换。外部地址变换的目的是要找到辅存(磁盘存储器)的实地址,并且把需要访问的那一页或那一个程序段调入到主存储器中。

在操作系统中,通常把页面失效当作一种异常故障来处理。在页式虚拟存储器和段页式虚拟存储器中,由于页面是按固定大小划分的,因此,页面失效可能发生在一条指令的执行过程中。例如,对于按照字节编址的主存储器,有可能出现一条指令跨越页面存放的情况。如果前一页已经在主存储器中,而后一页还没有调入主存储器,则在取指令过程中就可能发生页面失效。同样,在取操作数,特别是取字符串、间接寻址及写回运算结果的过程中都可能发生页面失效。对于这种异常故障,处理机必须立即响应并且处理,否则正在执行的指令无法继续执行下去。

如何保存和恢复故障点的现场,使得在故障处理完成之后能正确返回到断点处继续执行程序,是保证虚拟存储器能否正常工作的一个关键问题。这与一般的子程序调用或程序中断等情况不同,因为在正常情况下都是当一条指令执行完成后才转入子程序或中断处理程序的。有三种方法可解决这个问题。一种方法是采用硬件的缓冲寄存器,把执行该指令时的故障现场全部保存到缓冲寄存器中,等页面失效处理完成之后,可以完整地恢复现场,从故障点继续执行还没有执行完的指令。第二种方法是只保存部分现场,如程序计数器和处理机状态字等,等页面失效处理完后,从头开始执行还没有执行完成的指令。第三种方法是采用指令预判技术,对于那些有可能发生跨页执行的指令,如字符串指令等。在指令执行之前,就做页面失效处理,等执行这条指令所需要的所有页面全部调入主存储器之后,才开始执行该条指令。

磁盘存储器的地址格式如图 3.24 所示。由于一台机器可能有多个磁盘,因此,首先要给出磁盘号,然后是一个磁盘内的柱面号、磁头号 and 块号。磁盘存储器每一个物理块的大

小是 512 字节。由于在页式虚拟存储器和段页式虚拟存储器中,每一页面的大小是固定的,通常是磁盘存储器物理块大小的整数倍,这个倍数在外部地址变换软件中是知道的。对于段式虚拟存储器,在段表中有一个字段是段长度,根据段长度和磁盘存储器物理块的大小就能很容易地计算出本次页面失效时需要调入的磁盘存储器的物理块数。因此,在进行外部地址变换时只要给出磁盘存储器的起始地址,就能把一整页或一整个程序段都调入主存储器中。

磁盘号	柱面号	磁头号	块号
-----	-----	-----	----

图 3.24 磁盘存储器的地址格式

外部地址变换的过程如图 3.25 所示。由于页面失效的概率非常低,一般只有 1‰左右,因此,外部地址变换通常用软件来实现。每一个用户程序都有一张外页表(之所以称为外页表是因为它是在外部地址变换中使用的,与在内部地址变换中使用的页表被称为内页表相对应)。虚拟地址空间中的每一个页面或每一个程序段,在外页表中都有对应的一个存储字。在每一个存储字中除了必须有磁盘存储器的地址之外,至少还应该包括一个装入位。

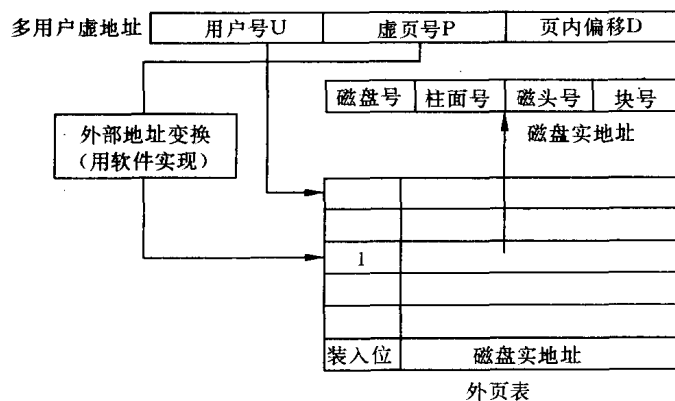


图 3.25 外部地址变换

由于每一个用户程序有一张外页表,因此,通过多用户虚地址中的用户号 U 就能够找到该用户程序的外页表起始地址。再通过虚页号 P 就能唯一确定外页表中与需要访问的页面相对应的那个存储字。如果该存储字中的装入位为“1”,则表示要访问的页面已经在磁盘存储器中,否则表示要访问的页面还不在于磁盘存储器中,需要从磁带、光盘存储器等海量存储器中调入。在段式虚拟存储器中,采用类似的方法,通过段号 S 唯一确定外页表中与需要访问的程序段相对应的那个存储字。

由于虚拟地址空间中的每一个页面(每一个程序段),在外页表中都要有对应的一个存储字,因此,外页表也要采用前面介绍的多级页表技术。

如果在整个程序的装入和执行过程中,不需要磁带、光盘等海量存储器介入,则外页表可以与内页表合并成一个页表。当装入位为“1”时,表示要访问的页面已经在主存储

器中,这时,页表中的地址字段指出该页在主存储器中的实页号。如果装入位为“0”,则表示要访问的页面在磁盘存储器中,这时,页表中的地址字段指出该页存放在磁盘存储器中的磁盘实地址。

3.2.3 加快内部地址变换的方法

在虚拟存储器中,如果不采取有效的措施,访问主存储器的速度将要降低几倍。这不符合存储系统的要求,因为在存储系统中,要求系统的访问速度接近于速度最高的那个存储器。造成虚拟存储器速度降低的主要原因有两个:

1. 在段式或页式虚拟存储器中,要访问主存储器必须先查段表或页表,在段页式虚拟存储器中,既要查段表也要查页表。如果段表和页表都在主存储器中,那么,包括访问主存储器本身这一次在内,主存储器的访问速度将要降低 2 至 3 倍。

2. 当页表和段表的容量超过了一个页面的大小时,它们就有可能被映象到主存储器的不连续的页面位置上。这样,按照地址查找主存实页号的方法,即把页表起始地址与多用户虚地址中虚页号相加,就不能成立。

目前的解决办法是采用多级页表。首先由页表基地址寄存器指出第一级页表的基地址,再用第一级页表各单元中的地址字段指出第二级页表的基地址,如此下去,构成一个树型结构的多级页表。在最后一级页表中给出主存实页号等信息。

如果虚拟存储空间的大小为 N_v ,页面的大小为 N_p ,一个页表存储字的大小为 N_d ,则页表的级数 g 可以通过下面的公式计算:

$$g = \left\lceil \frac{\log_2 N_v - \log_2 N_p}{\log_2 N_p - \log_2 N_d} \right\rceil$$

如果一个页式虚拟存储器的 $N_v = 4\text{GB}$, $N_p = 1\text{KB}$, $N_d = 4\text{B}$,以字节为单位,把这些数代入上面的公式,计算得到页表的级数:

$$g = \left\lceil \frac{\log_2 4\text{G} - \log_2 1\text{K}}{\log_2 1\text{K} - \log_2 4} \right\rceil = \left\lceil \frac{32 - 10}{10 - 2} \right\rceil = 3$$

即必须采用三级页表。第一级页表为 1 页,存储容量 1KB,可以有 256 个存储字(每个存储字占用 4 个字节),在这里只需要用其中的 64 个存储字。用第一级页表的 64 个存储字指向第二级页表的 64 个页面,每个页面各有 256 个存储字,这样,二级页表共有 16K 个存储字。同样,第三级页表共有 16K 个页面,即 4M 个存储字。这 4M 个存储字正好用来存放虚拟存储空间的 4M 个页面信息。因此,第一级页表为 1 个页面,第二级页表有 64 个页面,第三级页表则有 16K 个页面。

为了节省宝贵的主存储器资源,通常除一级页表必须驻留在主存储器中之外,二级和三级页表只需要驻留一小部分。只需把目前正在运行中的程序的相关页表,或者把已经调入到主存储器中的程序的相关页表驻留在主存储器中,绝大部分页表可以放在磁盘存储器中。当一个用户程序被调入主存储器时,再把相关的页表也同时装入主存储器中,并且修改页表中的装入位和主存地址等字段。

采用多级页表(包括段表)使得访问主存储器的次数又要增加。例如,对于一个采用三级页表的页式虚拟存储器,要想从主存储器中取出一个数据,必须依次查三次页表才能得

到这个数据在主存储器中的物理地址。包括读取这个数据本身在内,共需要访问四次主存储器。

要想使虚拟存储器的速度接近主存储器的速度,或者说,要想使虚拟存储器能够真正实用,必须加快查表的速度,

3.2.3.1 目录表

目录表的基本思想是:压缩页表的存储容量,用一个容量比较小的高速存储器来存放页表,从而加快页表的查表速度。

通常,多用户虚存空间要比主存储器的实存空间大得多。那么,在页式和段页式虚拟存储器中,虚存页面数就比实存页面数多得多。从页表中看,装入位为“1”(表示相应的页面在主存储器中)的页面存储字所占的比例很小。

有一种压缩页表的方法。页表只为已经装入到主存储器中的那些页面建立虚页号与实页号之间的对应关系。页表的每一个存储字中主要包括多用户虚页号、主存实页号、修改位和其他标志(如访问方式)等,不再需要装入位,并且采用相联方式访问。因此,把这种目录表称为相联目录表,简称目录表。

采用目录表法的虚拟存储器,其地址变换过程如图 3.26 所示。为了把多用户虚页号变换成主存实页号,要把多用户虚地址中的多用户虚页号(U 与 P 拼接起来)与相联存储器中的多用户虚页号字段逐个进行比较。如果有相等的,表示要访问的这个页面已经装入到主存储器中了。这时,读出该单元中的其他字段,其中,实页号字段中存放的就是与多用户虚页号相对应的主存实页号。只要把这个实页号 p 与多用户虚地址中的页内偏移 D 直接拼接起来就成了要访问的主存实地址。如果没有相等的,则表示要访问的那个页面还没有装入到主存储器中,这时,发出页面失效请求,从磁盘存储器中把要访问的那一个页面调入主存储器。

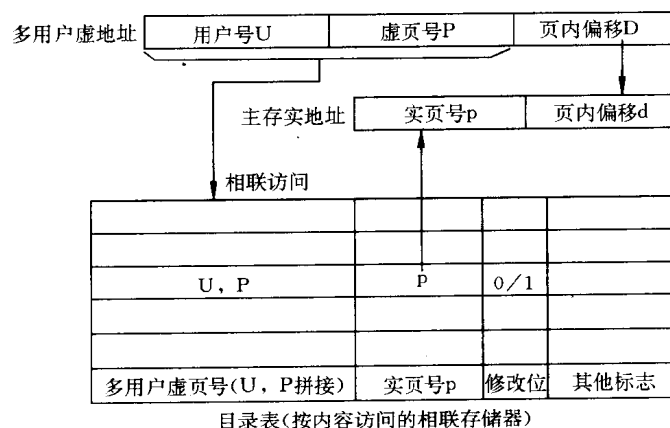


图 3.26 目录表法的地址变换过程

由于目录表是采用高速度小容量存储器实现的,与把页表放在主存储器中的方法相比,查表的速度要快得多。

然而,随着主存储器容量的增加,目录表的容量也必须增加。因此,目录表法的可扩展性比较差。当主存储器的容量增加到一定数量后,目录表的造价就会很高,查表的速度也会降低。

3.2.3.2 快慢表

由于程序在执行过程中具有局部性,因此,对页表中各存储字的访问并不是完全随机的。也就是说,在一段时间内,对页表的访问只是局限在少数几个存储字内。根据这一特点,可大大缩小目录表的存储容量。例如,容量为 8 个~16 个存储字,访问速度与 CPU 中的通用寄存器相当。这个小容量的页表称为快表,快表采用与目录表相同的相联方式访问。当快表中查不到时,再从存放在主存储器中的页表中查找实页号。与快表相对应,存放在主存储器中的页表称为慢表。慢表是一个全表,快表只是慢表的一个部分副本,而且只存放了慢表中很少的一部分。

实际上,快表与慢表也构成了一个由两级存储器组成的存储系统。与虚拟存储器和 Cache 存储器类似。在这个快、慢表的存储系统中,访问速度接近于快表的速度,存储容量是慢表的容量。

快表在英文资料中称为 TLB(translation lookaside buffer),有些中文资料中翻译成地址变换后行缓冲器,或地址转换后备缓冲存储器等。

由快慢表构成的虚拟存储器的地址变换过程如图 3.27 所示。用多用户虚页号同时去查快表和慢表。由于快表的查表速度要比慢表快得多,如果在快表中查到与多用户虚地址相等的存储字,就立即终止慢表的查表过程,并读出该存储字中的实页号 p 送入到主存储器的地址寄存器中。如果在快表中没有查到,就到存放在主存储器中的慢表中去。如果在慢表中查到了,则把查到的实页号 p 送入主存储器的地址寄存器,同时,也把这个实页号连同多用户虚地址等信息送入快表中。这时,若快表已满,则要采用某种替换算法,替换掉其中一个不常用的存储字。

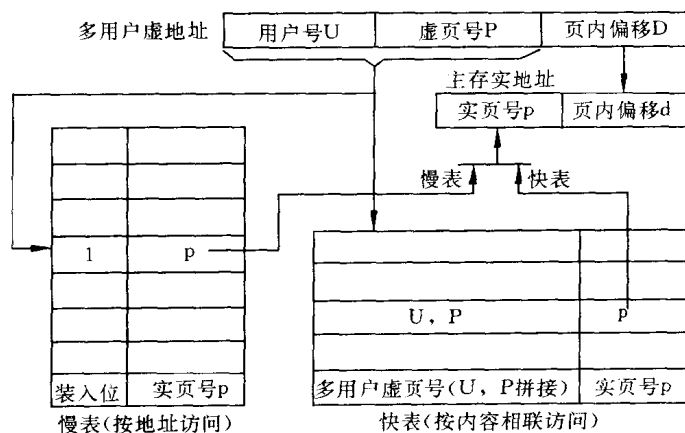


图 3.27 采用快慢表的地址变换过程

由于快表的查表速度非常快,与主存储器的一个存储周期相比几乎可以忽略不计。因此,只要快表的命中率很高,那么,虚拟存储器的访问速度就能与主存储器的工作速度很接近。

3.2.3.3 散列函数

在采用快慢表结构的虚拟存储器中,要提高快表的命中率,最直接的办法是增加快表的容量。快表的容量越大,命中率就越高。但是,由于快表是按相联方式访问的,当快表的容量增加时,它的查表速度就会降低。那么,能不能让快表不采用相联方式访问,而采用普通的按地址来访问呢?

如果要在一个按地址访问的存储器中查找一个信息,可以使用顺序查找法、对分查找法和散列查找法等。其中,散列(hashing)查找方法的速度最快。对于快表来说,就是要把多用户虚页号 P_v 变换成快表的地址 A_h 。函数关系是:

$$A_h = H(P_v)$$

散列函数的种类很多。由于快表中的散列函数必须用硬件来实现,因此,通常采用一些简单的函数关系。图 3.28 是一种折叠按位加散列函数,把 15 位多用户虚地址 P_v (内容)散列变换成 6 位的快表地址 A_h 。

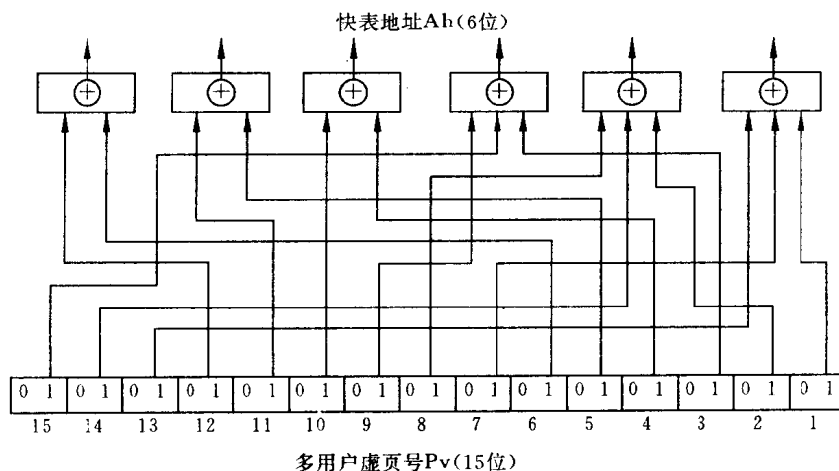


图 3.28 一种用硬件实现的散列函数

由于把一个大的多用户虚页号 P_v 散列变换成了一个小的快表地址 A_h ,因此,必然会有很多个多用户虚页号 P_v 都散列变换到相同的快表地址 A_h 中,这种现象称为散列冲突。例如,在图 3.28 中,平均会有 $2^{15} \div 2^6 = 2^9$,即 512 个多用户虚页号全部被散列变换成同一个快表地址。

为了避免因散列冲突而发生查快表时的错误,必须把多用户虚页号也加入到快表中去,并且与主存实页号存放在同一个快表存储字中。另外,还要用一个比较器,把从快表中读出来的多用户虚页号与多用户虚地址中的多用户虚页号进行比较。

采用散列变换实现快表按地址访问的虚拟存储器如图 3.29 所示。地址变换的过程是

这样的：首先把多用户虚地址中的多用户虚页号 P_v (由 U 和 P 拼接而形成) 送入硬件的散列变换部件，经散列变换后得到快表的地址 A_h 。然后用地址 A_h 访问快表，读出主存实页号 p 和多用户虚页号 P_v' 。把主存实页号 p 送入主存储器的地址寄存器，与页内偏移 d 直接拼接起来形成主存地址，并且用这个地址立即去访问主存储器。同时，把读出的多用户虚页号 P_v' 与多用户虚地址中多用户虚页号 P_v 在相等比较器中进行比较。如果比较结果相等，就继续刚才的访问主存储器的操作，否则，立即终止访问主存储器的操作。比较结果不相等表示发生了散列冲突，这时，需要去查存放主存储器中的慢表。查慢表的方法和快表的替换方法与上面介绍的采用相联存储器做快表的虚拟存储器相同。

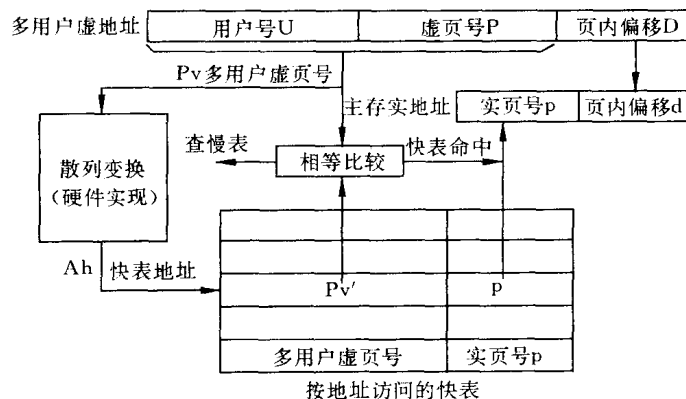


图 3.29 采用散列变换实现快表按地址访问

由于快表按地址访问，在保证访问速度的前提下，其存储容量可以比用相联存储器实现的快表大很多倍。虽然也有一次相等比较，但相等比较可以与访问存储器的操作同时进行。因此，这种快表按地址访问的快慢表结构，与上面介绍的采用相联存储器做快表的快慢表结构相比，快表的命中率要高很多，而且查表的速度也很快。

3.2.3.4 虚拟存储器举例

下面，介绍两个在机器中实际使用的虚拟存储器。

例 3.1 IMB370/168 计算机的虚拟存储器快表结构及地址变换过程

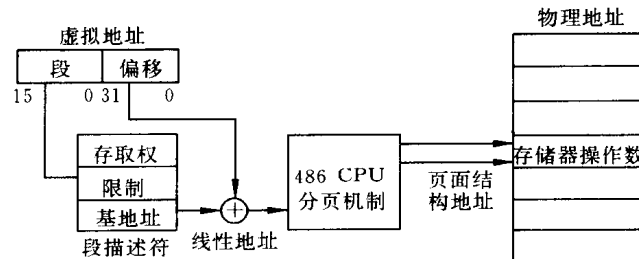
如图 3.30 所示，IMB370/168 计算机采用页式虚拟存储器，虚拟地址共长 36 位。其中，页面大小为 4K 字节，占 12 位。每个用户 (或每道程序) 最多允许占用 4K 个页面，因此，虚页号也占 12 位。最多允许 16G 个用户 (或 16G 道程序) 同时上机，用户号占 24 位，但是，实际上在一段时间内同时上机的用户数 (或程序道数) 一般不超过 6 个。

快表按地址访问，快表的地址由多用户虚页号经硬件的散列变换部件压缩后生成。快表地址共 6 位，因此，快表容量为 64 个存储字。

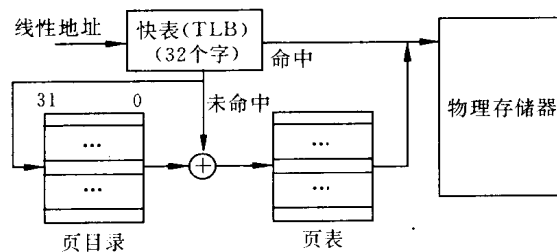
IMB370/168 计算机的虚拟存储器采用了两项新的措施。一项是在快表的每一个存储字中存放两对多用户虚页号与主存实页号，并采用两个相等比较器。只要其中有一个比较器的比较结果相等，就认为快表命中。这时，立即把命中的实页号 p 送到主存地址寄存

方式采用 4 级保护,使线性地址从 4GB 扩展到 64TB。在实方式下,最大存储容量是 1MB。在保护方式下可以运行 8086、80286 及 80386 处理机的所有软件。每个段的长度可以从 1 个字节到它的最大物理存储容量 4GB。

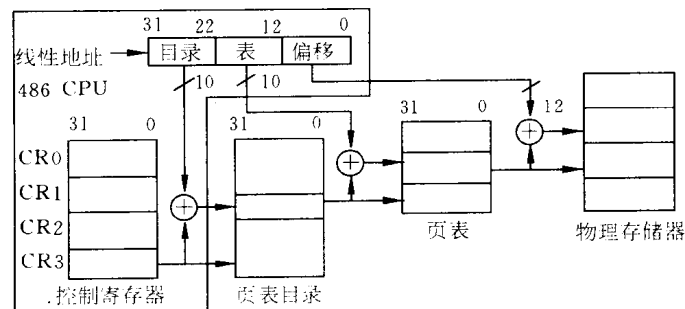
段可以在任意的基地址上开始,并允许段间有存储重叠。如图 3.31(a)所示,虚拟地址有一个 16 位的段选择器,用以确定由 i486 分页系统使用的线性地址空间的基地址。



(a) 分段产生线性地址



(b) 快表(TLB)操作



(c) 二级分页方案

图 3.31 Intel i486 CPU 的分页与分段机制

32 位的偏移用来指定一个段的内部地址。段描述符除了用来指出段的起始地址之外,还用来指定段的长度和该段的存取方式等。

分页特性在 i486 中是任选的,可以通过软件来控制它。当允许分页时,虚拟地址首先被转换成由虚页号和页内偏移组成的线性地址,然后再转换成主存物理地址。当禁止分页时,线性地址和主存物理地址是相同的。当选择的段为 4GB 时,整个主存储器就成为一个大段,这相当于分段机制实际上已经被禁止。因此,i486 处理机可以采用四种不同的方式

来组织它的存储器系统：页式虚拟存储器、段式虚拟存储器、段页式虚拟存储器及纯物理寻址的主存储器。

如图 3.31(b)表示,快表(TLB)可以将线性地址直接变换成主存物理地址,不需要像图 3.30(b)那样采用两级页表。

i486 的页面大小是 4KB。图 3.31(c)中的 4 个控制寄存器用来选择不同的页表目录。页表目录的大小为 4KB,共有 1 024 个页表存储字。第二级页表,每个页表的大小也是 4KB,能保存 1 024 个页面的信息。快表和两级页表的命中率取决于具体的程序和所采用的页面替换算法。以前的观察结果表明,快表的命中率在 98%左右。

3.2.4 页面替换算法及其实现

在虚拟存储器中,当发生页面失效时,需要从磁盘存储器中调入一页(或一段)到主存储器中。在页式和段页式虚拟存储器中,多用户虚页数比主存储器的实页数要多得多。在段式虚拟存储器中,虚存空间中能容纳的程序段数要比主存空间中能存放的相同长度的程序段数多得多。因此,必然会出现当主存中所有页面都已经被占用,或者所有主存空间都已经被占用,而又要从磁盘存储器中调入新页(或新段)的情况。这时,必须从主存储器中淘汰掉一个不常用的页面(或程序段),以便腾出主存空间来存放新调入的页面(或程序段)。那么,按照什么样的规则替换主存储器中的页面(或程序段)呢?这就是页面替换算法要解决的问题。

以下为了叙述方便,主要介绍页式和段页式虚拟存储器中的页面替换算法及其实现方法,在这两种虚拟存储器中都是以页面为单位进行调度的。而段式虚拟存储器是以程序段为单位进行调度的,但是它所采用的替换算法及算法的实现方法都是相同的。

评价一个页面替换算法好坏的标准主要有两个,一是命中率要高,二是算法要容易实现。要提高一个页面替换算法的命中率,首先要使这种算法能正确反映程序的局部性,其次是这种算法要能够充分利用主存中页面调度情况的历史信息,或者能够预测主存中将要发生的页面调度情况。

页面替换算法主要用在如下几个地方:

1. 虚拟存储器中,主存页面(或程序段)的替换。一般用软件实现。
2. Cache 中的块替换。一般用硬件实现。
3. 虚拟存储器的快慢表中,快表存储字(行)的替换。用硬件实现。
4. 虚拟存储器中,用户基地址寄存器的替换。用硬件实现。
5. 有些虚拟存储器中,目录表的替换。

3.2.4.1 页面替换算法

常用的页面替换算法有如下几种:

1. 随机算法,即 RAND 算法(random algorithm)。利用软件或硬件的随机数发生器来确定主存储器中被替换的页面。这种算法最简单,而且容易实现。但是,这种算法完全没有利用主存储器中页面调度情况的历史信息,也没有反映程序的局部性,所以命中率比较低。

2. 先进先出算法,即 FIFO 算法(first-in first-out algorithm)。这种算法选择最先调入主存储器的页面作为被替换的页面。它的优点是容易实现,能够利用主存储器中页面调度情况的历史信息,但是,没有反映程序的局部性。因为最先调入主存的页面,很可能也是经常要使用的页面。

3. 近期最少使用算法,即 LRU 算法(least recently used algorithm)。这种算法选择近期最少访问的页面作为被替换的页面。显然,这是一种非常合理的算法,因为到目前为止最少使用的页面,很可能也是将来最少访问的页面。该算法既充分利用了主存中页面调度情况的历史信息,又正确反映了程序的局部性。但是,这种算法实现起来非常困难。它要为每个页面设置一个很长的计数器,并且要选择一个固定的时钟为每个计数器定时计数。在选择被替换页面时,要从所有计数器中找出一个计数值最大的计数器。因此,通常采用另外一种变通的办法,就是下面的 LFU 算法。

4. 最久没有使用算法,即 LFU 算法(least frequently used algorithm)。这种算法把近期最久没有被访问过的页面作为被替换的页面。它把 LRU 算法中要记录数量上的“多”与“少”简化成判断“有”与“无”,因此,实现起来比较容易。LFU 算法的具体实现方法将在下一节中介绍。

5. 最优替换算法,即 OPT 算法(optimal replacemant algorithm)。上面介绍的几种页面替换算法主要是以主存储器中页面调度情况的历史信息为依据的,它假设将来主存储器中的页面调度情况与过去一段时间内的情况是相同的。显然,这种假设不总是正确的。最好的算法应该是选择将来最久不被访问的页面作为被替换的页面。这种替换算法的命中率一定是最高的。这就是最优替换算法,简称 OPT 算法。

要实现 OPT 算法,唯一的办法是让程序先执行一遍,记录下实际的页地址流情况。根据这个页地址流才能找出当前要被替换的页面。显然,这样做是不现实的。因此,OPT 算法只是一种理想化的算法,然而,它也是一种很有用的算法。实际上,经常把这种算法用来作为评价其他页面替换算法好坏的标准。在其他条件相同的情况下,哪一种页面替换算法的命中率与 OPT 算法最接近,那么,它就是一种比较好的页面替换算法。

下面,用一个典型的页地址流来评价 FIFO、LFU 和 OPT 三种页面替换算法的性能。其中,最主要的是命中率。当然,命中率还与程序的页地址流情况、页面的大小、主存储器的容量等因素有关。这些将在下一节介绍。

例 3.3 一个程序共有 5 个页面,分别为 P1~P5。程序执行过程中的页地址流(即程序执行中依次用到的页面)如下:

P1,P2,P1,P5,P4,P1,P3,P4,P2,P4

假设分配给这个程序的主存储器共有 3 个页面。图 3.32 表示 FIFO、LFU 和 OPT 三种页面替换算法对这 3 页主存的使用情况,包括调入、替换和命中等。图中,用“*”号标记下次将要被替换掉的页面。

在上面介绍的 5 种页面替换算法中,随机算法的命中率比较低,一般仅用于必须用硬件实现、而且对命中率要求不太高的场合。LRU 算法由于其实现起来特别困难,目前很少被采用。因此,在虚拟存储器中,实际上有可能被采用的页面替换算法就只有 FIFO 和 LFU 两种。

时间 t	1	2	3	4	5	6	7	8	9	10	实际命中次数
页地址流	P1	P2	P1	P5	P4	P1	P3	P4	P2	P4	
先进先出算法 (FIFO 算法)	1	1	1	1*	4	4	4*	4*	2	2	2 次
		2	2	2	2*	1	1	1	1*	4	
				5	5	5*	3	3	3	3*	
	调入	调入	命中	调入	替换	替换	替换	命中	替换	替换	
最久没有使用算法 (LFU 算法)	1	1	1	1	1	1	1	1*	2	2	4 次
		2	2	2*	4	4	4*	4	4	4	
				5	5*	5*	3	3	3*	3*	
	调入	调入	命中	调入	替换	命中	替换	命中	替换	命中	
最优替换算法 (OPT 算法)	1	1	1	1	1	1*	3*	3*	3	3	5 次
		2	2	2	2*	2	2	2	2	2	
				5*	4	4	4	4	4	4	
	调入	调入	命中	调入	替换	命中	替换	命中	命中	命中	

图 3.32 三种页面替换算法对同一个页地址流的调度过程

从图 3.32 中可以看出, FIFO 算法的命中率最低, LFU 算法的命中率与 OPT 算法很接近。这一结论具有普遍意义。因此, 在实际使用中, LFU 算法是一种比较好的算法。目前, 许多机器的虚拟存储器都采用 LFU 算法。

在页面调度中, 有一个需要特别注意的问题, 即所谓的“颠簸”(thrashing)现象。看下面的例子。

例 3.4 一个循环程序, 依次使用 P1, P2, P3, P4 四个页面, 分配给这个程序的主存页面数为 3 个。FIFO、LFU 和 OPT 三种页面替换算法对主存页面的调度情况如图 3.33 所示。OPT 算法命中 3 次, 而 FIFO 和 LFU 算法一次也没有命中。在 FIFO 和 LFU 算法中, 总是发生下次就要使用的页面本次被替换出去了。这就是“颠簸”现象。

一般来说, 对于一个循环程序, 当分配给它的页面数小于程序所需要的页面数时, 有可能发生“颠簸”现象。这时, 无论是 FIFO 算法, 还是 LFU 算法, 它们的命中率都明显地低于 OPT 算法。

对于例 2, 只要再多分配给它一个页面, 三种算法的命中率都能达到 4 次(最大值)。因此, 命中率不仅与页地址流有关, 而且也与分配给程序的主存页面数等有关。一般来说, 分配给程序的主存页面数越多, 虚页装入到主存到中的机会也就越多, 因此命中率也可能越高, 至少不应该下降。那么, 这一推测能否成立呢? 下面将介绍这一问题。

3.2.4.2 堆栈型替换算法

这里所说的堆栈型替换算法不是指某一种算法, 更不是指先进先出或先进后出的堆栈本身, 而是指一类算法。

在主存页面的分配和调度中, 影响命中率的因素很多, 这给页面分配和调度造成了困难。那么, 能否减少一些因素, 或者把某些因素简化一些呢?

时间 t	1	2	3	4	5	6	7	8	实际命中次数
页地址流	P1	P2	P3	P4	P1	P2	P3	P4	
先进先出算法 (FIFO 算法)	1	1	1*	4	4	4*	3	3	0 次
		2	2	2*	1	1	1*	4	
			3	3	3*	2	2	2*	
	调入	调入	调入	替换	替换	替换	替换	替换	
最久没有使用算法 (LFU 算法)	1	1	1*	4	4	4*	3	3	0 次
		2	2	2*	1	1	1*	4	
			3	3	3*	2	2	2*	
	调入	调入	调入	替换	替换	替换	替换	替换	
最优替换算法 (OPT 算法)	1	1	1	1	1*	1	1	1	3 次
		2	2	2	2	2*	3*	3	
			3*	4*	4	4	4	4*	
	调入	调入	调入	替换	命中	命中	替换	命中	

图 3.33 页面调度中的颠簸现象

堆栈型算法就是要研究影响命中率的一个主要因素,即分配给程序的主存页面数的增加与命中率的关系。弄清了这个问题的,不需要做大量的模拟工作,就能知道如何为程序分配主存页面数。

那么,什么是堆栈型替换算法呢? 它的定义如下:

对任意一个程序的页地址流作两次主存页面数分配,分别分配 m 个主存页面和 n 个主存页面,并且有 $m \leq n$ 。如果在任何时刻 t ,主存页面数集合 Bt 都满足关系:

$$Bt(m) \subseteq Bt(n)$$

则这类算法称为堆栈型替换算法。

简单地讲,堆栈型算法的基本思想是:随着分配给程序的主存页面数增加,主存的命中率也提高,至少不下降。

对于 LFU 算法和 LRU 算法,由于在主存中保留的是最近使用过的页面,如果先给某一个程序分配 n 个主存页面,那么在 t 时刻,这 n 个主存页面都是最近使用过的页面。如果再给这个程序多分配一个主存页面,那么在 t 时刻,这 $n+1$ 个主存页面也都是最近使用过的页面。因此,在这 $n+1$ 个主存页面中必然包含了前面的 n 个主存页面。所以,LFU 算法和 LRU 算法都是堆栈型算法。很显然,OPT 算法也是堆栈型算法。那么,FIFO 算法是不是堆栈型算法呢? 请看图 3.34 的情况。

在图 3.34 中,对于同一个页地址流,当分配给这个程序的主存页面数从 3 页增加到 4 页时,命中率反而从 3 次下降到 2 次。因此,FIFO 算法不是堆栈型算法。

由于堆栈型替换算法的命中率随分配给该程序的主存页面增加而单调上升,因此,在多道程序系统中,可以采用一种被称为页面失效频率法(PFF: page fault frequency)的动态页面调度方法。具体做法是:根据各道程序在实际运行过程中页面失效率的情况,由操作系统动态调整分配给每道程序的主存页面数。当一道程序的命中率低于某个限定值时

就增加分配给该道程序的主存页面数,以提高它的命中率。而当命中率高于某个限定值时就减少分配给该道程序的主存页面数,把节省出来的主存页面分配给其他程序。从而使整个系统的总的命中率和主存利用率都得到提高。

时间 t	1	2	3	4	5	6	7	8	9	10	11	12	实际命中次数
页地址流	P1	P2	P3	P4	P1	P2	P5	P1	P2	P3	P4	P5	
主存页面数 $n=3$	1	1	1*	4	4	4*	5	5	5	5	5*	5*	3次
		2	2	2*	1	1	1*	1*	1*	3	3	3	
			3	3	3*	2	2	2	2	2*	4	4	
	调入	调入	调入	替换	替换	替换	替换	命中	命中	替换	替换	命中	
主存页面数 $n=4$	1	1	1	1	1*	1*	5	5	5	5*	4	4	2次
		2	2	2	2	2*	2*	1	1	1	1*	5	
			3	3	3	3	3	3*	2	2	2	2*	
				4	4	4	4	4	4*	3	3	3	
	调入	调入	调入	调入	命中	命中	替换	替换	替换	替换	替换	替换	

图 3.34 FIFO 算法在主存页面数增加时命中率反而下降

3.2.4.3 页面替换算法的实现

虚拟存储器中的主存页面替换算法一般用软件实现。操作系统为了实现对主存储器的管理,设置有一个主存页面表。页面表中每一个存储单元记录主存储器中一个页面的当前使用状况。显然,主存页面表并不是前面介绍的那个页表。页表是用来保存虚页号与主存实页号之间的映象关系,并且在程序运行过程中实现地址变换用的,它是面向用户程序空间的,每一个用户,或者每一道程序都有一张页表。而主存页面表是面向主存储器的,整个主存储器只有一张主存页面表。主存页面表存放在主存储器中,它所记录的主要内容如图 3.35 所示。

实页号	占用位	程序号	段页号	使用位 (计数器)	程序优先级	历史位 Hb (未使用计数器)	其他信息
0							
1							
·							
·							
·							
P-1							

图 3.35 主存页面表的结构(P 为主存页面数)

因为主存页面表中的每一个存储字对应于主存储器中的一个页面,而且按顺序存放的,因此实页号这一字段可以省略。占用位用来表示哪些页面已经被占用。占用位为“0”表示该页面是空的,没有被占用,占用位为“1”则表示该页面已经被占用。程序号和段

页号表示该页面是被哪个程序的哪个段、哪个页占用。

为了实现 FIFO 替换算法,主存页面表中使用位字段应该设置成一个计数器。每当有一个页面装入主存储器时,就让该页面的计数器清零,而其他已经装入到主存储器中的页面所对应的计数器加“1”。在需要替换时,只要找出计数器的值最大的页面就是最先进入主存储器的页面。

实现 LFU 算法也相当容易。在一般情况下,只需要一个使用位即可。开始时,所有页面的使用位全为“0”。当一个页面中的任何一个存储单元被访问后,就把该页面的使用位置为“1”。这样,当主存储器的所有页面都已经被占用,即所有占用位均为“1”时,又有新的页面需要调入主存储器时,就要进行替换。替换的方法很简单,只要找出使用位为“0”的页面作为被替换的页面即可。

如果所有页面的使用位已经为“1”,又要调入新页时,就没有办法确定要被替换的页面。因此,采用使用位来实现 LFU 算法时,不允许所有页面的使用位全部为“1”的情况发生。有以下几个办法可以解决这个问题。

一种简单的办法是,当所有页面的使用位都为“1”时,就立即把所有页面的使用位全部清成“0”。因此,LFU 算法中的这个“近期”就是指从上次所有使用位全部清为“0”到这次使用位全部为“1”这一段时间。在一般情况下,这一段时间是不固定的。这种方法很简单,而且容易实现。

第二种办法是定期法。每间隔一个固定的时间,如几个毫秒或几秒,把所有页面的使用位全部清为“0”。那么,这时候的“近期”就是一个固定的时间间隔。当然,这个固定的时间间隔要选择得恰当。既要保证不发生全部使用位都为“1”的情况,又要使这个时间间隔尽可能长些。

另一种比较好的办法是,在主存页面表中再设置一个历史位 Hb,或者叫做“未使用计数器”。每间隔一段时间扫描所有页面的使用位。如果使用位为“0”,则把对应的 Hb 加“1”,否则,把 Hb 清为“0”。同时,无论使用位原来是“0”还是“1”,都把它们清为“0”。这样,扫描结束时,所有页面的使用位就都是“0”,相当于又开始了一个“近期”。这种方法与上面两种方法的不同处在于,它把每一个“近期”都记录在 Hb 中了。因此,Hb 的值越大,说明所对应的页面最久没有被访问过,就应该成为最先被替换掉的页面。实质上,前两种方法只反映了一个“近期”内主存页面的使用情况,而后一种方法能反映多个“近期”内主存页面的使用情况。

3.2.5 提高主存命中率的方法

在本章第一部分中,曾经给出一个存储系统的等效访问速度的计算公式:

$$T = H \cdot T_1 + (1 - H) \cdot T_2$$

在虚拟存储器中, T_1 是指主存储器的访问周期,其中包括查页表和段表等所需要的时间。在本章第三节中已经介绍了用目录表、快慢表和散列函数等方法来缩短查表的时间,并且介绍了提高快表命中率的一些方法。当快表的命中率很高时,查页表和段表所花的时间与主存储器的存储周期相比几乎可以忽略不计。因此,公式中的 T_1 就与主存储器访问周期很接近。而 T_2 是指磁盘存储器的访问速度。由于当主存储器不命中时,一般用软件来调

度,因此, T_2 通常是很大的。

从上面的分析中可以看出,要提高虚拟存储器的等效访问速度,提高主存的命中率是一个非常关键的因素。

影响主存命中率的主要因素有如下几个:

1. 程序在执行过程中的页地址流分布情况;
2. 所采用的页面替换算法;
3. 页面大小;
4. 主存储器的容量;
5. 所采用的页面调度方法。

在这些因素中,页地址流的分布情况是由程序本身决定的,系统设计人员一般无能为力。页面替换算法在上一节中已经介绍过。目前,多数机器都采用 LFU 算法,它是一种堆栈型算法。在当前看来,已经是一种比较好的算法了。下面,对影响主存命中率的另外三个因素作简单的分析。

3.2.5.1 页面大小的选择

页面大小与主存命中率的关系不是线性的,页面大小还与主存储器的存储容量和程序的页地址流分布情况等多种因素有关。在图 3.36 中给出页面大小 S_p 、主存容量 S 与命中率 H 的关系曲线。

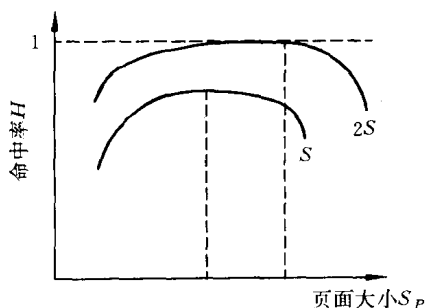


图 3.36 页面大小与主存命中率的关系

从图 3.36 中可以看出,在分配给程序的主存容量 S 一定的情况下,当页面大小 S_p 比较小的时候,命中率 H 随页面大小 S_p 的增大而逐渐提高。当页面大小增加到某个值时,命中率达到最大。然后,随着页面大小的增加,命中率反而降低。另外,从两条曲线的比较中可以看出,当分配给程序的主存容量 S 增大时,命中率能普遍提高,关于命中率与主存容量的关系,在下面的一小节中还要介绍。

图 3.36 的曲线关系可以这样来解释:在程序执行过程中,假设 A_i 和 A_{i+1} 是两次相邻的访问主存储器的逻辑地址, $d = |A_i - A_{i+1}|$ 。如果 $d < S_p$, 那么,随着 S_p 的增大, A_i 和 A_{i+1} 在同一个页面内的可能性就会增加,即 H 随着 S_p 的增大而提高。如果 $d > S_p$, 那么, A_i 和 A_{i+1} 一定不在同一个页面内。随着 S_p 的增大,在分配给该程序的主存容量一定的情况下,主存的页面数就要减少,页面的替换将更加频繁。这样, A_i 和 A_{i+1} 两个地址所在的页面都在主存储器中的可能性就会减少,即 H 随着 S_p 的增大而降低。当 S_p 比较小的时候,前一种情况是主要的,因此, H 随着 S_p 的增大而提高。当 S_p 达到某一个最大值之后,后一种情况成为主要的,因此, H 随着 S_p 的增大而降低。

在设计一个虚拟存储器时,页面大小的选择一般要通过对典型程序的模拟实验来确定。早期的许多计算机系统,页面大小一般为 1KB。目前,大多数机器的页面大小都取 4KB、8KB 或 16KB。

另外,当页面大小增大时,由于每个程序或程序段的最后一个页面一般是装不满的,由此造成的浪费也要增加。相反,当页面大小减小时,页表(指慢表)和页面表在主存储器中所占的比例将增加。这两种情况都要降低主存储器的利用率。因此,页面大小的选择要综合考虑多方面的因素。

3.2.5.2 主存容量

主存命中率 H 随着分配给该程序的主存容量 S 的增加而单调上升,如图 3.37 所示。在 S 比较小的时候, H 提高得非常快。随着 S 的逐渐增加, H 提高的速度逐渐降低。当 S 增加到某一个值之后, H 几乎不再提高。

在页面替换算法中有这样一个结论,对于堆栈型算法,命中率随着分配给程序的页面数的增加而提高。当分配给程序的主存容量增加时,如果页面大小是一定的,那么,页面数就会增加,因此,命中率也将提高。如果不是堆栈型算法,命中率虽然不会单调上升,在局部可能有下降,但总的趋势还是上升的。

从图 3.37 中可以得到这样一个启发,在为一道程序分配主存空间时,对主存命中率的要求不能过分。当主存容量增加到某一个值之后,命中率提高得非常慢。这时,主存储器中不活跃部分所占的比例很大,主存资源的利用率就会很低。

实际上,操作系统在为程序分配主存空间时,是以页为单位分配的。因此,图 3.37 所示的不应该是一条平滑的曲线,而是台阶型的。在分配给程序的主存容量比较小的时候,台阶非常陡,随着主存容量的增加,台阶越来越平缓。

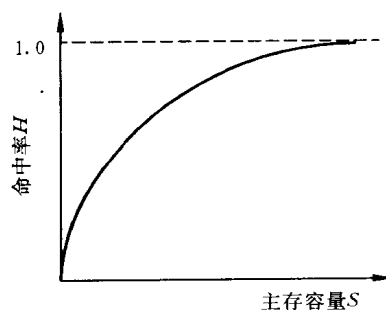


图 3.37 主存命中率 H 与主存容量 S 的关系

3.2.5.3 页面调度方式

在操作系统中,页面调度通常有两种方式。一种是分页式,这种方式在程序装入主存储器之前就对程序进行链接装配,并且要在把整个程序都调入到主存储器中之后才能开始运行。另一种方式是请求页式,这种方式只在发生页面失效时,才把要访问的页面进行链接装配,并调入到主存储器中。

前一种方式的主存命中率可以达到 100%。但是,主存利用率比较低,这是因为主存储器中不活跃部分所占的比例比较大。而且,当主存剩余空间小于程序所需要的主存空间时,这个程序将无法装入到主存储器中运行。

目前,大多数机器采用请求页式调度方式。这种方式虽然具有主存利用率高(只要主存储器还剩余有一个页面,程序就能调入到主存储器中开始运行)等优点,但是,在程序执行过程中经常要发生页面失效,而且处理页面失效需要比较长的时间。特别在程序刚开始运行时,页面失效很频繁。从前面的图 3.32 中也可以看出,无论采用什么样的页面替换算法,在程序开始执行时的一段时间内,都在不断地调入页面,主存命中率很低。只有当调入

的页面数比较多时,命中率才开始上升。因此,就产生了另外一种折中的调度方式,即所谓预取式调度方式。

预取式调度方式根据程序的局部性特点,在程序被挂起之后又重新开始运行之前,先把上次停止运行前一段时间内用到的页面先调入到主存储器,然后才开始运行程序。这样,在程序一开始运行时,主存储器中就已经装入了一定数量的页面,从而可以避免在程序刚开始运行时,频繁发生页面失效的情况。当然,这种调度策略也不一定总是有效的。如果调入的页面在程序开始运行后用不上,那么,它不仅在调入这些页面的过程中浪费了时间,而且还占用了主存资源。

3.3 高速缓冲存储器(Cache)

目前,高性能处理机的工作主频都在 200MHz 以上,有的已经达到 500MHz,而且普遍采用超标量和超流水线等技术,在一个周期内能并行执行多条指令。然而,这些指令和指令执行过程中所需要的数据都来自主存储器。一般主存储器采用动态存储器(DRAM)实现,其存储周期仅为 100ns 到 200ns。假设处理机在每个周期平均执行 2 条指令,每条指令平均访问(读操作数和写结果)2 个数据。如果指令和数据都是存放在主存储器中的话,那么,处理机每个周期就要访问主存储器 6 次。只要简单地计算一下就会发现,处理机需要主存储器的速度和主存储器实际能提供的速度两者相差在 100 倍左右。

在第二章中,我们已经看到,在处理机中通常有几十个通用寄存器。存放在这些通用寄存器中的数据尽管也来自主存储器,但是,它们可以被重复使用,因此,缓解了主存储器的部分压力。第五章,还将介绍指令和数据的先行缓冲存储器。它们也能缓解主存储器的部分压力。一般认为,采用这些缓冲技术之后,可以把高性能处理机与主存储器之间的速度差缩小到 30 倍左右。目前,这 30 倍左右的差距是依靠 Cache 来填补的。

一般处理机中有一级 Cache,它与主存储器构成一个两级的存储系统。一些高性能处理机都采用两级 Cache。其中,第一级在 CPU 内部,它的容量比较小,速度很快。第二级在主板上,容量比较大,速度比第一级要低 5 倍左右。也有部分高性能处理机采用三级 Cache,前两级都在 CPU 内部。本节主要介绍由一级 Cache 与主存储器构成的 Cache 存储系统(以下简称 Cache)。

Cache 与上面介绍的虚拟存储器有许多相似的地方,如地址变换方法,替换算法等,但也有许多不同的地方。具体可看表 3.4。

因为 Cache 是全部用硬件来调度,因此,它不仅对应用程序员是透明的,而且对系统程序员也是透明的。在一般情况下,无论是应用程序员还是系统程序员都看不到系统中有 Cache 存在,更不知道是采用了几级 Cache。在他们的感觉中,程序是放在主存储器中的,或者是放在容量比主存储器大得多,访问速度和编址方式等与主存储器完全一样的虚拟存储器中的。

目前,Cache 一般用高速静态存储器(SRAM)实现,存储周期为几十毫微秒,存储容量在几十个千字节至几兆字节之间,价格比较贵。Cache 与主存储器之间的速度差在 3 倍~10 倍。如果速度差太大,要采用多级 Cache。

表 3.4 Cache 与虚拟存储器的主要区别

存储系统	Cache	虚拟存储器
要达到的目标	提高(主存)速度	扩大(主存)容量
实现方法	全部硬件	软件为主,硬件为辅
两级存储器速度比	3 倍~10 倍	10^5 倍
页(块)大小	1 字~16 字	1KB~16KB
等效存储容量	主存储器	虚拟存储器
透明性	对系统和应用程序员	仅对应用程序员
不命中时处理方式	等待主存储器	任务切换

Cache 与主存储器之间以块为单位进行数据交换。块的大小通常以在主存储器的一个存储周期中能访问到的数据长度为限。如果主存储器采用并行或低位交叉存取方式,在一个主存周期内可以访问到多个字。例如,IBM370/168 计算机的主存储器采用 4 个存储体低位交叉存取方式,每个存储字的字长是 8 个字节,因此,每块的大小为 32 个字节。这样,当 Cache 不命中时,只要等待一个主存周期,CPU 就能得到所需要的数据。为了做到这一点,在主存储器中,还必须把 Cache 的访问请求安排在最高优先级。

3.3.1 基本工作原理

在 Cache 存储系统中,把 Cache 和主存储器都划分成相同大小的块。因此,主存地址由块号 B 和块内地址 W 两部分组成。同样,Cache 的地址也由块号 b 和块内地址 w 组成。Cache 的基本工作原理如图 3.38 所示。

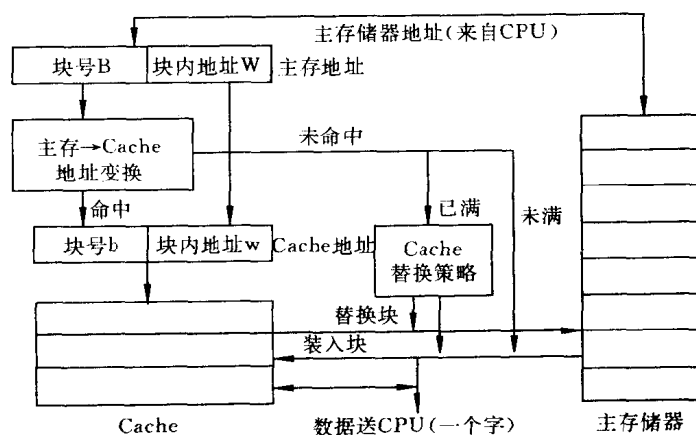


图 3.38 Cache 工作原理

当 CPU 要访问 Cache 时,CPU 送来主存地址放入主存地址寄存器中。通过主存-Cache 地址变换部件把主存地址中的块号 B 变换成 Cache 的块号 b 放入 Cache 地址寄存

器中,并且把主存地址中的块内地址 W 直接作为 Cache 的块内地址 w 装入到 Cache 地址寄存器中。如果变换成功(称为 Cache 命中),就用所得到的 Cache 地址去访问 Cache,从 Cache 中取出数据送往 CPU。如果变换不成功,则产生 Cache 失效信息,并且用主存地址访问主存储器。从主存储器中读出一个字送往 CPU,同时,把包括被访问字在内的一整块都从主存储器中读出来,装入到 Cache 中去。这时,如果 Cache 已满,则要采用某种 Cache 替换算法把不常用的一块先调入主存储器中原来存放它的地方,以便腾出空间来存放新调入的块。由于程序具有局部性特点,每次块失效时都把一块(由多个字组成)调入到 Cache 中,能够提高 Cache 的命中率。

通常,Cache 的容量比较小,主存储器的容量要比它大得多。那么,Cache 中的块与主存储器中的块是按照什么样的规则建立对应关系的呢?在这种对应关系下,主存地址又是如何变换成 Cache 地址的呢?这就是下一节要解决的问题。

3.3.2 地址映象与变换方法

在 Cache 中,地址映象是指把主存地址空间映象到 Cache 地址空间,具体地说,就是把存放在主存中的程序按照某种规则装入到 Cache 中,并建立主存地址与 Cache 地址之间的对应关系。而地址变换则是指当程序已经装入到 Cache 之后,在实际运行过程中,把主存地址如何变换成 Cache 地址。

地址的映象和变换是紧密相关的。采用什么样的地址映象方法,就必然有与这种映象方法相对应的地址变换方法,因此,必须把它们放在一起介绍。

无论采用什么样的地址映象方式和地址变换方式,都要把主存和 Cache 划分成同样大小的存储单位,每个存储单位称为“块”。在进行地址映象和地址变换过程中,都以块为单位进行调度。

在选取地址映象方法时,要考虑多方面的因素。如地址变换的硬件是否容易实现,地址变换的速度是否快,这种映象方法的主存空间利用率是否高,以及在把主存中的一块装入到 Cache 中时,发生块冲突的概率如何等。

根据所采用的地址映象方法和地址变换方法的不同,有多种不同类型的 Cache。下面,将介绍 5 种地址映象和地址变换的方法。

3.3.2.1 全相联映象及其变换

全相联映象方式是指主存中的任意一块可以映象到 Cache 中的任意一块的位置上,如图 3.39 所示。如果 Cache 的块容量为 C_b ,主存的块容量为 M_b ,则主存与 Cache 之间的映象关系共有 $C_b \times M_b$ 种。如果采用目录表来存放这种映象关系,则目录表的容量为 C_b ,字长为 Cache 地址中的块号长度与主存地址中的块号长度之和再加一个有效位。

采用全相联映象方式的地址变换过程如图 3.40 所示。主存块号与 Cache 块号的映象关系存放在用一个高速存储器实现的目录表中。目录表的每一个存储字由三部分组成:主存块号、Cache 块号及一个有效位。

在程序执行过程中,当要访问 Cache 时,用主存地址中的块号 B 与目录表中的主存块号字段进行相联比较。如果发现有相等的,表示要访问的数据已经装入到 Cache 中了,

这种情况称为 Cache 命中。这时,只要把目录表中与主存块号字段在同一个存储字中的 Cache 块号读出来,并把它与主存地址中的块内地址 w 直接拼接起来,就得到 Cache 地址。用这个 Cache 地址去访问 Cache,把读出来的一个字送往 CPU 就完成了访问过程。如果在相联比较中没有发现相等的,表示要访问的那个块还没有装入到 Cache 中,这种情况称为 Cache 没有命中,或称为 Cache 失效。这时,要用主存地址去访问主存储器,把从主存储器中读出来的一个字送往 CPU。同时,把包括被访问字在内的一块都从主存储器中读出来装入到 Cache 中,另外,还要修改目录表中的主存块号字段,把当前的主存块号 B 写到目录表的这个存储字中。

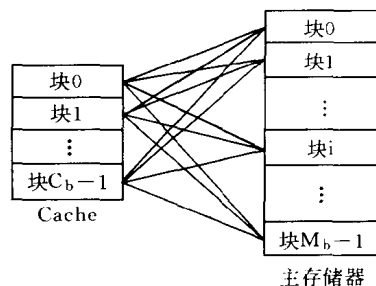


图 3.39 全相联映象方式

有效位是用来标记目录表中的各个存储字是否有效的。如果有效位为“1”,表示目录表中由主存块号 B 与 Cache 块号 b 建立的映象关系是有效的。实际上也表示在 Cache 的第 b 块中存放的数据是主存第 B 块中数据的正确副本。如果有效位为“0”,表示目录表中的主存块号 B 与 Cache 块号 b 之间的映象关系是无效的,或者说它们之间根本没有什么关系,Cache 的第 b 块中存放的数据也不是主存第 B 块中数据的正确副本。因此,在访问目录表时,也要看有效位的状态。在 Cache 失效时,也要对有效位的状态进行管理。具体方法与直接映象方式的地址变换过程相同。

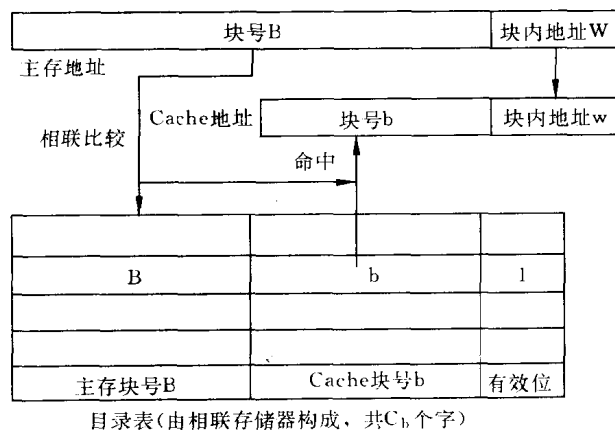


图 3.40 全相联地址变换

采用全相联映象和变换方式最突出的优点是块的冲突率最小,Cache 的利用率也最高。但是,需要一个相联访问速度很快、容量为 C_b 的相联存储器,其代价很高。而且,相联比较所花费的时间将影响 Cache 的访问速度。

在虚拟存储器中,一般都采用全相联映象方式。这是因为当发生主存页面失效时,必须进入异常故障处理,由软件完成地址变换,即把多用户虚地址变换成主存实地址,而且要从磁盘存储器调入一个页面装入到主存储器中。由于磁盘存储器的寻址是由机械动作来完成的,相对速度很慢。因此,在一般多用户和多任务系统中,当发生主存页面失效

时,通过任务切换,切换到一个程序已经装入到主存储器中的任务中去执行。由于这些原因,在虚拟存储器中,采用全相联映象方式是必然的。

3.3.2.2 直接映象及其变换

直接映象方式是一种最简单,也是最直接的方法。主存中一块只能映象到 Cache 的一个特定的块中。假设主存的块号为 B , Cache 的块号为 b ,则它们之间的映象关系可以用下面的公式表示:

$$b = B \bmod C_b$$

其中, C_b 是 Cache 的块容量。另外, M_b 是主存的块容量, M_e 是主存的区容量。

直接映象方法的映象关系如图 3.41 所示。把主存按 Cache 的大小分成区,在一般情况下,主存的容量是 Cache 容量的整倍数。主存每一个分区内的块数与 Cache 的总块数正好相等。直接映象方式只能把主存各个区中相对块号相同的那些块映象到 Cache 中同一块号的那个特定块中。例如,主存的块 0 只能映象到 Cache 的块 0 中,主存的块 1 只能映象到 Cache 的块 1 中。同样,主存区 1 中的块 C_b (在区 1 中的相对块号是 0) 也只能映象到 Cache 的块 0 中。

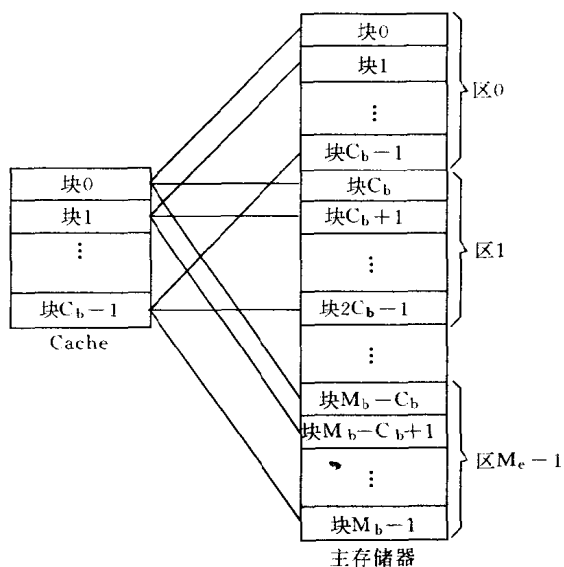


图 3.41 直接相联映象方式

根据上面给出的地址映象规则,整个 Cache 地址与主存地址的低位部分是完全相同的。在图 3.42 中,主存地址中的块号 B 与 Cache 地址中的块号 b 是完全相同的。同样,主存地址中的字内地址 W 与 Cache 地址中的块内地址 w 也是完全相同的。主存地址比 Cache 地址长出来的部分称为区号 E 。

在程序执行过程中,当要访问 Cache 时,为了实现主存块号到 Cache 块号的变换,需要有一个存放主存区号的小容量存储器。这个存储器的容量与 Cache 的块数相等,字长为

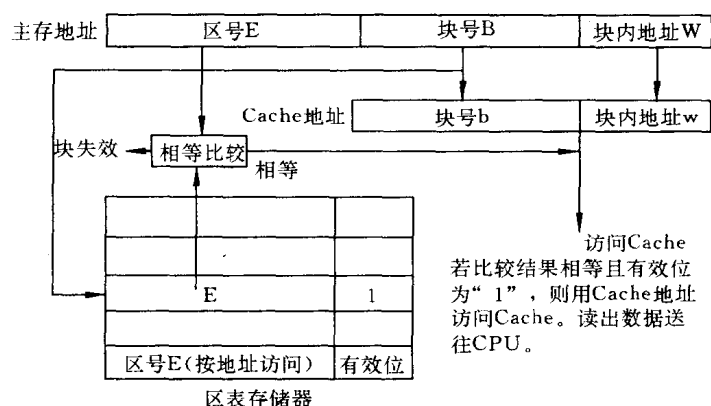


图 3.42 直接相联地址变换

主存地址中区号 E 的长度,另外再加一个有效位。

直接映象方式的地址变换过程如图 3.42 所示。首先用主存地址中的块号 B 去访问区号存储器(按地址访问)。把读出来的区号与主存地址中的区号 E 进行比较,根据比较结果和与区号在同一存储字中的有效位情况作如下处理。

如果区号比较结果相等,有效位为“1”,则 Cache 命中。表示要访问的那一块已经装入到 Cache 中了,这时的 Cache 地址(与主存地址的低位部分完全相同)是正确的。用这个 Cache 地址去访问 Cache,把读出来的数据送往 CPU。

其他情况均为 Cache 没有命中,或称为 Cache 失效。表示要访问的那个块还没有装入到 Cache 中,这时,要用主存地址去访问主存储器,把读出来的一个字送往 CPU。同时,把包括被访问字在内的一块都从主存储器中读出来。

如果区号比较结果相等,有效位为“0”,表示 Cache 中的这一块已经被作废。这时,只要把从主存中读出来的新块按照 Cache 的块地址装入到 Cache 中,并且把有效位置“1”即可。

如果区号比较结果不相等,有效位为“0”,表示 Cache 的这一块是空的。这时,把从主存中读出来的新块装入到 Cache 中,并把有效位置“1”。另外还要把主存区号写到区号存储器的相应单元中。

如果区号比较结果不相等,有效位为“1”,表示原来在 Cache 中存放的那一块是有用的。这时,必须先把 Cache 中的这一块写回到主存储器中原来存放它的存储单元中,腾出地方来,才能把从主存中读出的新块装入到 Cache 中。另外也要把主存区号写到区号存储器的相应单元中。

为了提高 Cache 的访问速度,可以把图 3.42 中的区号存储器与 Cache 合并成一个存储器,如图 3.43 所示。这样,用主存地址中的块号 B 访问 Cache,把区号和这一块的所有数据同时都读出来。由于 Cache 中一块的字数很少(1 至 16 个),通过一个多路选择器,在块内地址的控制下,把所需要的那个字选出来送往 CPU。如果 Cache 没有命中,处理方法与图 3.42 中的相似。

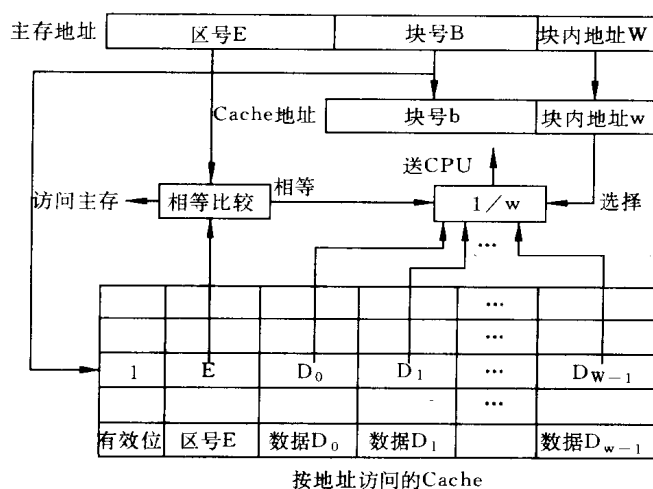


图 3.43 快速度的直接相联地址变换

图 3.43 所示的 Cache, 它的访问速度要比图 3.42 的快得多。多路选择器的实现也比较容易。在前面介绍的并行和交叉访问存储器中, 从多个存储体中读出的数据也采用多路选择器输出。两个多路选择器是完全相同的。

直接映象方法的优点是硬件实现很简单, 不需要采用相联访问的存储器, 访问速度也比较快。实际上, 采用直接相联方式的 Cache 不需要进行地址变换, 因为主存地址的低位部分就是 Cache 的地址。

直接映象方式的主要缺点是块的冲突率比较高。当主存中的两个或两个以上的块都映象到 Cache 的同一块中, 而这些块又都是当前的常用块时, Cache 的命中率会很低。这时, 即使 Cache 中还有很多空闲的块, 也帮不上忙。

IBM370/158 及 PDP-11/60 等计算机均采用直接映象方式。

3.3.2.3 组相联映象及其变换

组相联方式是目前在 Cache 中用得比较多的一种地址映象和变换方式。它是介于全相联和直接相联之间的一种折中方案。

组相联映象方式也采用与全相联映象方式和直接相联映象方式相同的方法把主存和 Cache 按同样大小划分成块。所不同的是: 组相联映象方式还把主存和 Cache 按同样大小划分成组, 每一组都由相同的块数组成, 如图 3.44 所示。

由于主存储器的容量要比 Cache 的容量大得多, 因此, 主存的组数要比 Cache 的组数多。从主存的组到 Cache 的组之间采用直接映象方式。在主存中的一组与 Cache 中的一组之间建立了直接映象关系之后, 在两个对应的组内部采用全相联映象方式。在图 3.44 中, 只画出了组间的直接映象关系。在有直接映象关系的任意一个主存的组与相应的 Cache 的组之间是全相联映象关系。例如, 主存的组 0 直接映象到 Cache 的组 0, 则主存中的块 0 可以映象到 Cache 中的块 0、块 1、...、块 $G_b - 1$ 中的任意一块中。在图 3.44 中,

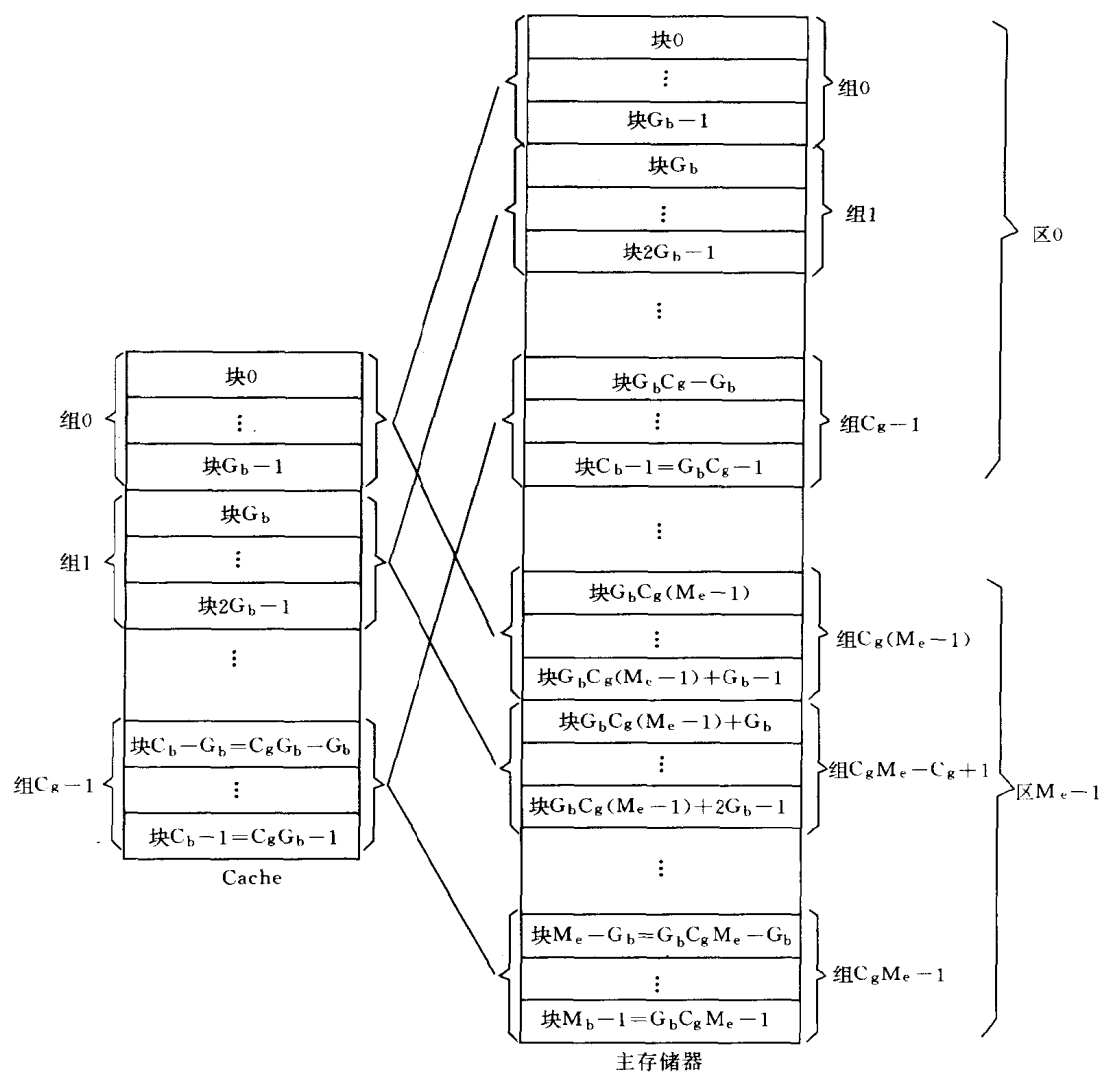


图 3.44 组相联映象方式

G_b 是每一组的块容量, C_g 是 Cache 的组容量。另外, 与直接映象方式一样, M_e 是主存的区容量, C_b 为 Cache 的块容量, M_b 为主存的块容量。

为了实现主存块号到 Cache 块号的变换, 需要一个由高速小容量存储器做成的块表存储器。块表存储器采用按地址访问和按相联访问两种方式工作。在块内采用相联方式访问, 在块之间采用按地址方式访问。块表的容量与 Cache 的块容量相等, 字长为主存地址中的区号 E 、组内块号 B 与 Cache 地址中的组内块号 b 的长度之和, 另外再加一个有效位及其他控制字段等。

组相联映象方式的地址变换过程如图 3.45 所示。在程序执行过程中, 当要访问 Cache 时, 用主存地址中的组号 G 按地址访问块表存储器。从块表存储器中读出来的不仅

仅是一个字,而是一组字,字的个数等于组内的块容量 G_b 。把这些字中的区号和块号与主存地址中相应的区号 E 和块号 B 进行相联比较。如果发现相等的,表示要访问的数据已经装入到 Cache 中了,这种情况称为 Cache 命中。这时,只要把同一个存储字中的 Cache 块号 b 读出来,并且把它与主存地址中直接送过来的组号 g 和块内地址 w 直接拼接起来,就得到 Cache 地址。用这个 Cache 地址去访问 Cache,把读出的一个字送往 CPU。如果在相联比较中没有发现相等的,表示要访问的那个块还没有装入到 Cache 中,这种情况称为 Cache 没有命中,或称为 Cache 失效。这时,要用主存地址去访问主存储器,把从主存储器中读出来的一个字送往 CPU。同时,把包括被访问字在内的一块都从主存中读出来装入到 Cache 中。另外,还要修改块表存储器等。

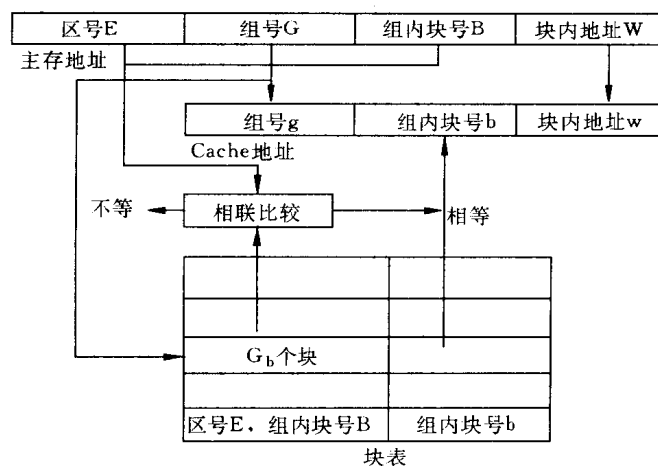


图 3.45 组相联映象方式的地址变换

有效位的使用和管理方法与直接映象方式中的地址变换过程完全相同,在图 3.45 中没有画出来。

为了提高 Cache 的访问速度,可以把 Cache 的地址变换与访问 Cache 并行进行,并采用流水线方式工作。

在组相联映象方式中,组内的块容量一般是很小的,如 4 块左右。因此,可以采用把块表存储器中一个相联比较的组按块方向展开存放,如图 3.46 所示。这样,可以用多个相等比较器来代替相联访问,以加快查表的速度。许多实用的组相联 Cache 都采用这种方法。它的地址变换过程与上面的方法类似。

另外,块表中的有效位是用来标记一个块的映象关系是否成立的。它的使用和管理方法与直接映象方式中的地址变换过程完全相同。

当一个块内的字数不多时,可以把块表与 Cache 合并成一个存储器。图 3.46 中的块号 b 字段用 Cache 中的数据字来代替,并且用 Cache 地址中块内地址部分控制一个多路选择器,从读出的多个数据字中选择一个所需要的数据字。

组相联映象方式与直接映象方式相比,最明显的优点是块的冲突概率大大降低。例如,在直接映象方式中,主存中的一块只能映象到 Cache 中的一个特定块中。而在每组为

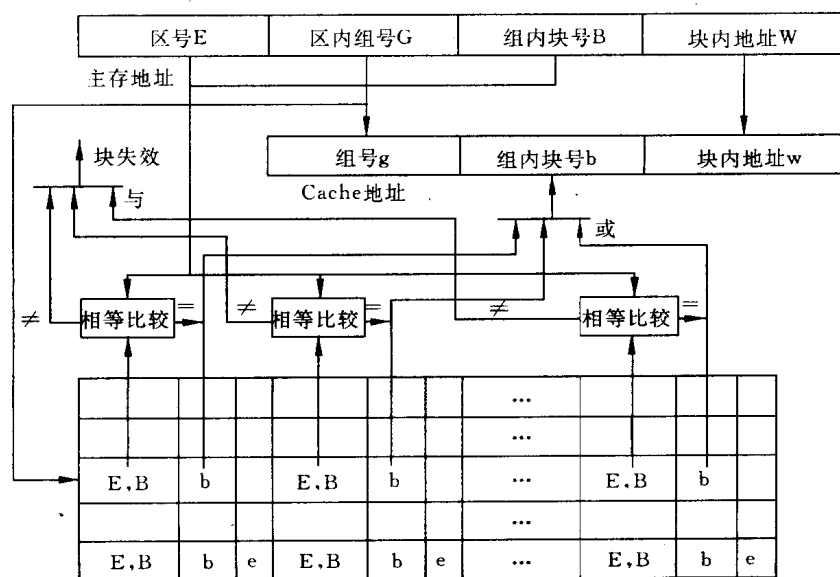


图 3.46 组相联地址变换的一种实现方法

4 块的组相联映象方式中,主存中的一块能够映象到 Cache 的 4 个特定的块中。因此,采用组相联映象方式,Cache 中块的利用率能够大幅度提高,Cache 的块失效率能够明显降低。但是,由于组相联映象方式在组内部需要进行相联比较,因此,实现的难度和造价要比直接映象方式高。

组相联映象方式与全相联映象方式相比,实现起来要容易得多,但 Cache 的命中率与全相联映象方式很接近。因此,组相联映象方式在许多机器中得到广泛的应用。

在组相联映象方式中,当每组的块容量 G_b 为 1 时,就成了直接映象方式。同样,当每组的块容量 G_b 与 Cache 的块容量 C_b 相等时,就成了全相联映象方式。因此,直接相联映象方式和全相联映象方式是组相联映象方式的两个极端情况。

在 Cache 的容量和每块的大小确定之后,可以通过选择每组的块容量 G_b 和 Cache 的组容量 C_g ,即分配 Cache 地址中组号 g 和组内块号 b 两个字段的长度,来优化 Cache 的性能。主要包括块冲突概率,块的失效率,查表的速度,实现的复杂性和成本等。一般来说,每组的块容量 G_b 越大,块的冲突概率和 Cache 的失效率就越低,但由于映象关系复杂,查表的速度就越慢,实现的成本也越高。

表 3.5 给出了采用组相联映象方式的一些典型机器的 Cache 分组情况。从表中可以看出,每组的块容量(组的大小)一般都很小。有近一半的机器选择每组为两块,最多的为 16 块。其主要原因是:当每个组的块容量增大时,需要进行相联访问的存储器的容量将增加,造成查表的速度降低,实现成本增加。但是,当每个组的块数太少时,块的冲突概率和 Cache 的失效率就会增大。例如,当每组的块容量为 2 时,主存的各个分区中相对块号相同的那些块只能映象到 Cache 的两个特定块中,当这些块中有两个以上是当前的常用块时,就会出现所谓的颠簸现象。

表 3.5 采用组相联映象方式的典型机器的 Cache 分组情况

机器型号	Cache 的块容量 C_b	每组的块容量 G_b	Cache 组容量 C_g
DEC VAX-11/780	1 024	2	512
Amdahl 470/V6	512	2	256
Intel i860 D-Cache	256	2	128
Honeywell 66/60	512	4	128
Amdahl 470/V7	2 048	4	512
IBM 370/168	1 024	8	128
IBM3033	1 024	16	64
Motolola 88110 I-Cache	256	2	128

组相联映象和变换方式有很多种变型,它们各有不同的特点。下面,介绍一种常用的组相联映象及变换方式,称为位选择组相联映象及变换方式,或位选择算法组相联映象及变换方式。

3.3.2.4 位选择组相联映象及其变换

位选择组相联映象方式的映象关系如图 3.47 所示。与上面的一般组相联映象方式相比,Cache 仍然分组,而主存不再分组。主存除了分块之外,还按照 Cache 的组容量 C_g 分区。主存每个分区中的块容量与 Cache 中的组容量相等。

主存中的块与 Cache 中的组之间是直接映象关系,而主存中的块与 Cache 中组内部的各个块之间是全相联映象方式。例如,主存中的块 0 与 Cache 中的组 0 之间是直接映象关系,而主存中的块 0 与 Cache 中组 0 内的所有块之间是全相联映象关系,即主存中的块 0 可以映象到 Cache 的块 0、块 1,直到块 G_b-1 中的任意一块中。

把位选择组相联映象方式与一般组相联映象方式进行比较,主存中的一块能够映象到 Cache 中的总的块数并没有变化,都等于 Cache 的组容量。所不同的是,在一般组相联映象方式中,主存中的一个组与 Cache 中的一个组之间是多个块到多个块的映象。而在位选择组相联映象方式中,改成了主存中的一个块到 Cache 中的一个组之间是一个块到多个块的映象。映象关系明显简单,实现起来可以容易些。另外,从数据的分布情况看。对于一般组相联映象方式,主存中的几个连续块映象到 Cache 中可能也是连续的,而对于位选择组相联映象方式,主存中的连续块映象到 Cache 中肯定是不连续的,它们被分散到 Cache 的各个组中。由于在主存与 Cache 之间是以块为单位进行调度的,而 Cache 是以字为单位访问的,只要 Cache 中的一个字不跨越两个块,在 Cache 内部的块与块之间的分布是否连续对 Cache 的正常工作是没有关系的。

位选择组相联地址变换方式的一种实现方法如图 3.48 所示。与上面的图 3.46 比较,它把主存地址中的组内块号字段与区号字段合并到一起,成了一个比较长的区号 E。由于主存每个区中的块容量与 Cache 中的组容量是相等的,而且它们之间采用直接映象方式,

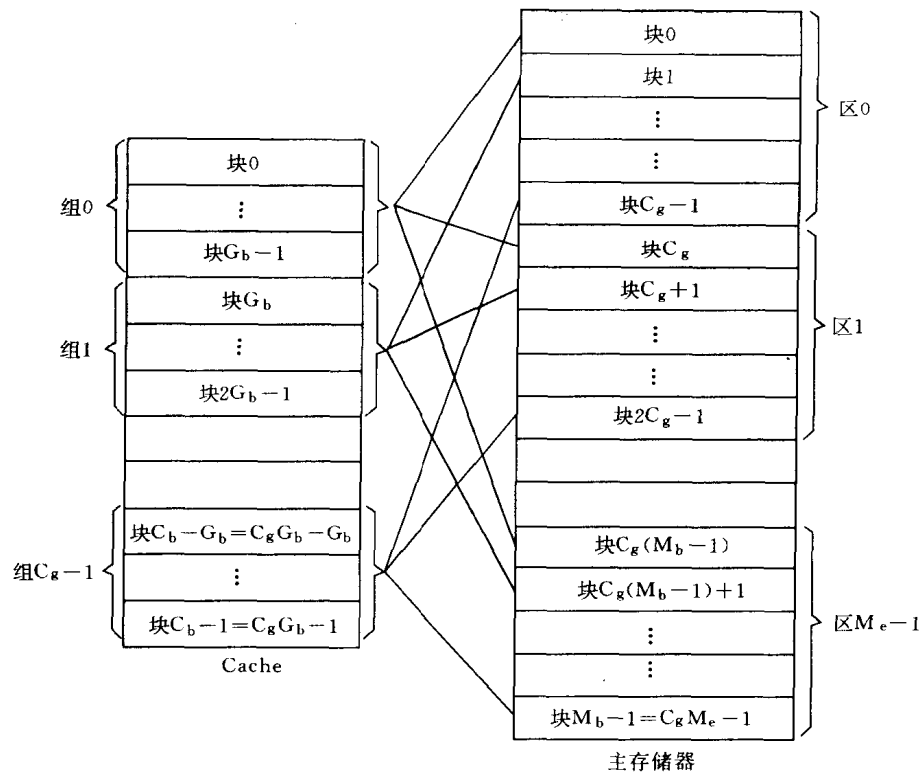


图 3.47 位选择组相联映象方式

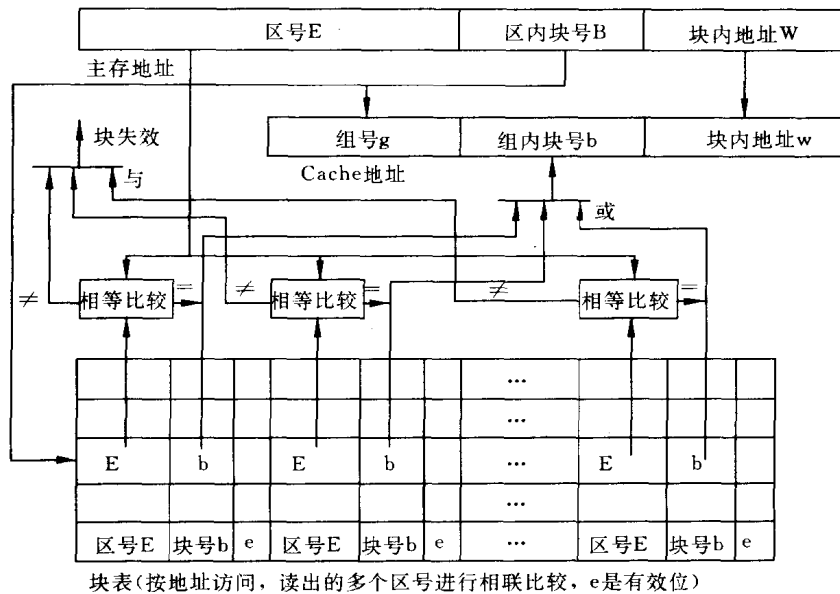


图 3.48 位选择组相联地址变换的一种实现方式

因此,主存地址中的区内块号 B 可以直接作为 Cache 地址中的组号 g 。同样,Cache 地址中的块内地址 w 可以直接从主存地址的块内地址 W 取得。整个地址变换过程的目的是要从块表中找到 Cache 的块号 b 。

另外,块表中的有效位是用来标记区号 E 与块号 b 建立的映象关系是否成立的。它的使用和管理方法可以参照直接映象方式。

位选择组相联的地址变换过程与一般组相联的地址变换过程基本相同。主要不同是,在块表中存放的和参与相联比较的只有主存地址中的区号 E ,而不再有组内块号 B ,使得实现起来稍容易了些。

3.3.2.5 段相联映象及其变换

把组相联映象方式中的映象关系改变一下,组内改用直接映象方式,而组间改用全相联映象方式,这样就成了段相联映象方式。实际上,段相联映象方式是组相联映象方式的一种变形,也是介于全相联与直接相联两种映象方式之间的一种折中方案。

段相联映象方式的映象关系如图 3.49 所示。它与上面介绍的所有映象方式一样,也把主存和 Cache 按同样大小划分成块。所不同的是:段相联映象方式还把主存和 Cache 都按同样大小划分成段。无论是 Cache 还是主存,每一段的块容量都相等。如果说,段相联映象方式中的段就相当于组相联映象方式中的组的话,那么,段相联映象方式还有两个与

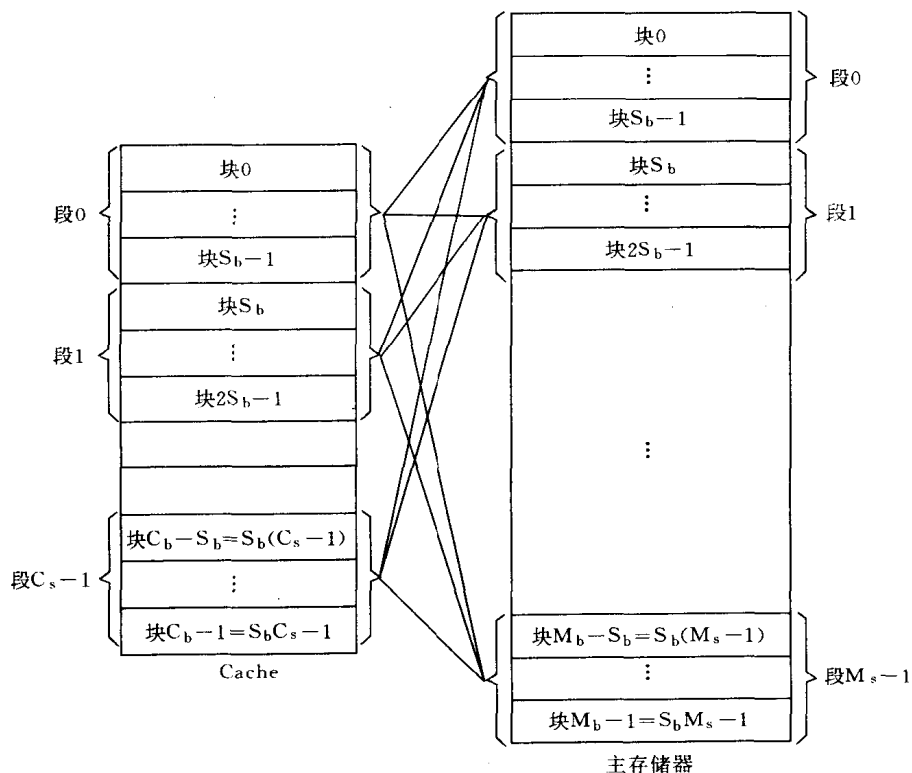


图 3.49 段相联映象方式

组相联映象方式根本不同的地方。其一：在段相联映象方式中，主存的段与 Cache 的段之间采用全相联映象方式，当在主存的一个段与 Cache 的一个段之间建立了映象关系之后，这两个段内部的块之间采用直接映象方式。其二：由于主存的段与 Cache 的段之间采用全相联映象方式，因此，主存就没有必要再分区了，这一点是与组相联映象方式根本不同的。因为这种映象方式与组相联映象方式有许多不同的地方，为了加以区别，把它称为段相联映象方式。

为了降低相联比较的复杂度，一般把 Cache 的段容量做得比较小，而每一个段内的块容量很大。这一点，也是与一般组相联映象方式不同的。

在图 3.49 中， C_s 表示 Cache 的段容量， M_s 表示主存的段容量， S_b 表示每一个段内的块容量，而 C_b 仍表示 Cache 的块容量。根据上述映象规则，主存中的块 0 可以映象到 Cache 的块 0、块 S_b 、块 $2S_b$ 、……、块 $S_b(C_s-1)$ 中。

段相联的地址变换过程如图 3.50 所示。主存的地址和 Cache 的地址都由三个部分组成，分别是段号、段内块号和块内地址。由于主存的容量要比 Cache 的容量大得多，因此，主存地址中的段号 S 要比 Cache 地址中的段号 s 长得多。而两个地址中的段内块号和块内地址是一样长的。另外，由于主存的段与 Cache 的对应段之间是采用直接映象方式的，因此，Cache 地址中的段内块号 b 可以从主存地址中的段内块号 B 直接拷贝过来。同样，Cache 地址中块内地址 w 也可以直接从主存地址中的块内地址 W 直接拷贝过来。Cache 地址中只有一个段号 s 需要从段表中查出来。

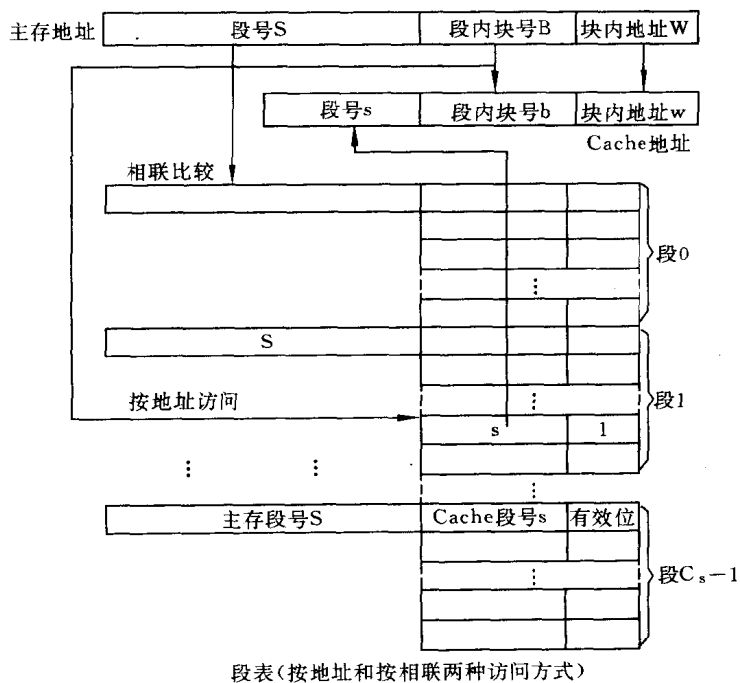


图 3.50 段相联地址变换方式

段表由主存段号 S 和 Cache 段号 s 组成,每一个主存段号 S 下有 S_b 个 Cache 段号 s 。由于 Cache 的段容量一般很小(不超过 16),因此,段表中长度比较长的主存的段号 S 很少,绝大部分是长度比较短的 Cache 的段号。

当需要访问 Cache 时,首先用主存地址中的段号 S 与段表中的主存段号字段进行相联比较。如果发现有相等的,再用主存地址中的段内块号 B 按地址访问相应段表中的 Cache 段号部分,读出 Cache 的段号 s 。最后把这个段号 s 与从主存地址中直接拷贝过来的段内块号 b 及块内地址 w 直接拼接起来就得到 Cache 地址。如果相联比较没有发现相等,则发出 Cache 块失效请求,需要访问主存储器。

另外,段表中与每一个 Cache 段号相联系的还有一个有效位,它被用来表示这个主存段号与 Cache 段号之间建立的映象关系是否有效,即已经存放在 Cache 中的这个块是否是主存的有效副本。当用主存地址中的段号 S 与段表中的主存段号字段进行相联比较时,如果没有发现相等的,则发生段失效,这时要把本段中所有块的有效位都清除,原来本段内各个块与主存建立的映象关系要全部撤消,以便与主存的一个新段重新建立映象关系。这一点是与上面介绍的地址变换方式不同的。有效位的其他使用及管理方法与前面介绍的地址变换方式相同。

采用段相联映象和变换方式的主要优点是段表比较简单,实现的成本相对较低。例如,一个容量为 256KB 的 Cache,分成 8 个段,每段 2 048 块,每块 16B。在段表存储器中只需要存储 8 个主存地址的段号 S 。而在上面所介绍的其他地址变换方式中,在块表(相当于段相联地址变换方式中段表)中要存储 $8 \times 2\,048 = 16\,384$ 个区号 E (相当于段相联地址变换方式中段号 S),两者相差两千多倍。当然,采用段相联映象方式的实现成本要低得多。

在段相联方式中,块的冲突概率和 Cache 的失效率与 Cache 的段容量 C_s 有关。Cache 划分的段数越多,主存中某一块能够映象到 Cache 中的块数也越多,因此,块的冲突概率和 Cache 的失效率也就越低。这一点与前面介绍的组相联映象方式及位选择组相联映象方式有些类似,下一节将比较详细地讨论这个问题。

段相联映象及变换方式也有一个明显的缺点。当发生段失效,要把本段内各个块原来已经与主存建立起来的许多映象关系全部撤消。由于段相联映象方式中每一段的块容量比较大,因此,被作废的块数也比较多。

3.3.3 Cache 替换算法及其实现

在把主存地址变换成 Cache 地址的过程中,如果发现 Cache 块失效,则需要从主存中调入一个新块到 Cache 中。而来自主存中的这个新块往往可以装入到 Cache 中的多个块中。当可以装入这个新块的几个 Cache 块已经被装满时,就要使用 Cache 替换算法,从那些块中找出一个不常用的块,把它调回到主存中原来存放它的那个地方,腾出一个块来存放从主存中来的这个新块。

直接映象及变换方式实际上不需要替换算法,这是因为主存中的一块只能装入到 Cache 的唯一一个块中。如果 Cache 的这一块是空的,则可以装入,如果 Cache 的这一块已经被占用,唯一的办法就是把它替换出去。

在全相联映象及变换方式中,由于主存中的一块可以装入到 Cache 中任意一块的位置上,因此,它的替换算法也就最复杂。

在组相联和位选择组相联映象及地址变换方式中,需要从 Cache 同一组内的几个块中选择一块替换出去。

在虚拟存储器中,介绍了五种页面替换算法。其中只有最久没有使用算法(LFU)比较好。虚拟存储器中的页面替换算法主要是用软件实现的,而且虚拟存储器都采用全相联映象方式。而在 Cache 中,由于 Cache 的访问速度很高,替换算法必须全部用硬件实现,而且 Cache 中一般不采用全相联映象方式。因此,Cache 中的块替换算法与虚拟存储器中的页面替换算法虽然有一些相同的地方,但更主要的是不相同的地方。

Cache 替换算法中最简单的是随机法。例如,PDP-11/70 的 Cache 采用组相联映象方式,每组只有两块。当发生块冲突时,使用一个 2 态的随机数发生器,从组内的两块中任意选择一块替换出去。

随机法的优点是实现起来非常容易,因此,在有些小型微型计算机中被采用。它的缺点是既没有利用程序的局部性特点,也没有利用历史上的块地址流分布情况,因此,它的效果往往不好。

下面介绍四种 Cache 替换算法。其中,轮换法实际上是一种先进先出(FIFO)算法,另外三种实际上都属于最久没有被使用(LFU)算法,只是它们的实现方式各不相同。由于 Cache 的块替换算法与它的实现方法紧密相关,因此,把算法及该算法的实现方法放在一起介绍。

3.3.3.1 轮换法及其实现

轮换法通常用于组相联映象及地址变换方式中,常见的有两种实现方法。

1. 每块一个计数器

在上面介绍组相联及位选择组相联地址变换方式中,块表内用来表示每一块映象关系的一个存储字由三字段组成,包括主存地址的区号 E 和块号 B(在位选择组相联映象方式中只有主存地址的区号 E,没有块号 B)、Cache 的块号 b 及一个有效位 e。为了实现轮换法,还要再增加设置一个替换计数器字段。计数器字段的长度与 Cache 地址中的组内块号字段的长度相同。

替换方法及计数器的管理规则是:

(1) 被装入或被替换的块,它所属的计数器清为“0”,同组的其他块所属的计数器都加“1”。

(2) 需要替换时,在同组中选择计数器的值最大的块作为被替换的块。

下面举一个例子来说明轮换法是如何实现的。

Solar-16/65 机的 Cache 采用位选择组相联映象方式。Cache 每组的块容量为 4,因此,每块用一个 2 位的计数器。刚开始时,同组的 4 个计数器均为 00。装入及替换的过程见表 3.6。实际上,是装入还是替换是由有效位决定的,而替换计数器用来指示装入或替换的块号。

表 3.6 一种轮换法的装入及替换过程

序列	初始值	装入块 00	装入块 01	装入块 10	装入块 11	替换块 00
块 00	00	00	01	10	11	00
块 01	00	01	00	01	10	11
块 10	00	01	10	00	01	10
块 11	00	01	10	11	00	01

2. 每组一个计数器

分析上面的方法,实际上同组内的所有块是按顺序轮流替换的。为此,只要为每个组设置一个计数器即可。

替换规则和计数器的管理方法很简单。本组有替换时,计数器加“1”,计数器的值就是要被替换出去的块号。

例如,NOVA3 计算机的 Cache 采用组相联映象方式。Cache 每组的块容量为 8,每组设置一个 3 位计数器。在需要替换时,计数器的值加“1”,用计数器的值直接作为被替换块的块号。

轮换法的优点是实现比较简单。与随机法相比,它能够利用历史上的块地址流情况,把最先装入的块作为被替换的块。但是,与随机法相同,它也没有能够利用程序的局部性特点。因为最先装入 Cache 的块,很可能也是经常要使用的块。因此,它的效果虽然比随机法好,但仍不理想。

3.3.3.2 LFU 算法及其实现

最久没有使用算法(LFU 算法: least frequently used algorithm)在虚拟存储器中是使用最普遍的一种算法。这种算法选择最久没有被访问的块作为被替换的块,显然,这是一种比较合理的做法。因为到目前为止最久没有被访问的块,很可能也是将来最少访问的块。因此,LFU 算法既利用了历史上 Cache 中块地址流的调度情况,又正确反映了程序的局部性特点。

为了实现 LFU 算法,要在块表中为每一块设置一个计数器。计数器的长度与上面介绍的轮换法相同。

计数器的使用及管理规则是:

1. 被装入或被替换的块,其对应的计数器清为“0”,同组中其他所有块所属的计数器都加“1”。

2. 命中的块,其对应的计数器清为“0”。同组中其他所有计数器中,凡是计数器的值小于命中块所属计数器原来值的,都加“1”,其他计数器不变。

3. 需要替换时,在同组的所有计数器中选择计数值最大(一般为全 1)的计数器,它所对应的块就是要被替换的块。

下面举一个例子。

IBM 370/165 机的 Cache 采用组相联映象方式。每组有 4 块,为了实现 LFU 替换算

法,在块表中为每一块设置一个 2 位的计数器。在访问 Cache 的过程中,块的装入、替换及命中时,计数器的工作情况如表 3.7 所示。

表 3.7 LFU 替换算法的工作情况

块地址流	主存块 1		主存块 2		主存块 3		主存块 4		主存块 5		主存块 4	
	块号	计数器	块号	计数器	块号	计数器	块号	计数器	块号	计数器	块号	计数器
Cache 块 0	1	00	1	01	1	10	1	11	5	00	5	01
Cache 块 1		01	2	00	2	01	2	10	2	11	2	11
Cache 块 2		01		10	3	00	3	01	3	10	3	10
Cache 块 3		01		10		11	4	00	4	01	4	00
	装入		装入		装入		装入		替换		命中	

LFU 算法与前面介绍的两种轮换法相比,它的控制逻辑要复杂些,增加了判断和处理命中的情况。然而,由于它既能比较正确地利用程序的局部性特点,又能比较充分地利用历史上块地址流的分布情况。因此,LFU 算法的命中率是比较高的。在虚拟存储器的页面替换算法中,曾经说明过 LFU 算法是一种堆栈型算法。随着每一组的块容量增加,Cache 的命中率能够单调上升。

3.3.3.3 比较对法

比较对法实际上也是一种 LFU 算法。所不同的只是,它不用计数器来实现,而采用硬联逻辑实现。

LFU 算法实际上要把同一组内的各个块按照被访问过的时间顺序排序,从而找出最久没有被访问过的块。一个两态的器件(触发器)能够记录两个块之间的先后顺序,多个块之间的先后顺序可以用多个两态器件的组合来实现,从而可以在多个块中找出最久没有被访问过的那个块来。

下面以三个块为例,说明比较对法的实现方法。

假设三个块分别称为块 A、块 B、块 C,它们之间的组合共有 $C_3^2=3$ 种,分别为 AB、AC、BC。这些组合可以用 3 个两态的触发器来表示,用 T_{AB} 表示 B 块比 A 块更久没有被访问过,其他关系也可以用类似的方法表示。3 个块之间共有 6 种不同的排列。它们分别表示为: T_{AB} 、 T_{BC} 、 T_{AC} 、 $\overline{T_{AB}}$ 、 $\overline{T_{BC}}$ 、 $\overline{T_{AC}}$ 。

如果要表示块 C 最久没有被访问过,可以这样来表达:表示块 C 最久没有被访问过的三个块的排列顺序有两种可能,从最近到最远分别是:块 A、块 B、块 C 和块 B、块 A、块 C。根据逻辑关系,很容易写出块 C 最久没有被访问过表达式:

$$C_{LFU} = T_{AB} \cdot T_{BC} \cdot T_{AC} + \overline{T_{AB}} \cdot T_{BC} \cdot T_{AC} = T_{AC} \cdot T_{BC}$$

同样,也可以写出块 A 和块 B 最久没有被访问过表达式,分别为:

$$A_{LFU} = \overline{T_{AB}} \cdot \overline{T_{AC}}$$

$$B_{LFU} = T_{AB} \cdot \overline{T_{BC}}$$

用硬件实现的逻辑图如图 3.51 所示。

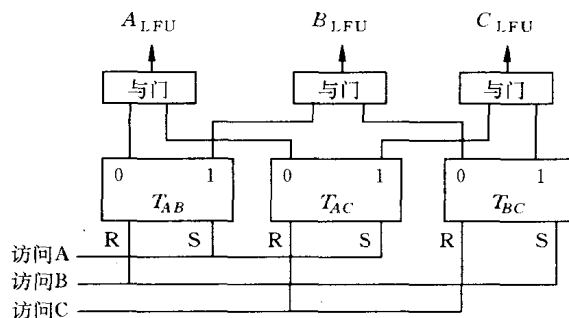


图 3.51 每组 3 个块的比较对法

在每次访问之后要改变比较对触发器的状态。例如,在访问块 A 之后,要把块 A 设置为最近被访问过,比较对触发器的状态应该为: $T_{AB}=1, T_{AC}=1$, 触发器 T_{BC} 的状态没有关系。同样,在访问块 B 之后,比较对触发器的状态应该为: $T_{AB}=0, T_{BC}=1$, 触发器 T_{AC} 的状态没有关系。在访问块 C 之后,比较对触发器的状态应该为: $T_{AC}=0, T_{BC}=0$, 触发器 T_{AB} 的状态没有关系。

从图 3.51 中可以看出,在每组 3 个块时,用比较对法实现块替换需要 3 个 R-S 型触发器和 3 个与门,每个与门需要两个输入端。当每组中的块数增多时,需要触发器的个数、与门的个数也要增加,与门的输入端的个数也要增加。

假设每组的块容量为 G_b ,则需要的触发器的个数可以由下面的公式计算:

$$C_{G_b}^2 = \frac{G_b \cdot (G_b - 1)}{2}$$

需要与门的个数为 G_b 个,每个与门的输入端个数为 G_b-1 个。随着每组中的块容量增加,所需要的触发器的个数及与门的个数成平方关系增加。表 3.8 给出每组不同块容量情况下,所需要的触发器的个数、与门个数及与门输入端的个数。

表 3.8 每组块容量与所需触发器、与门及与门输入端个数的关系

每组块容量	3	4	6	8	16	64	256
触发器个数	3	6	15	28	120	2 016	32 640
与门个数	3	4	6	8	16	64	256
与门输入端个数	2	3	5	7	15	63	255

从表 3.8 中可以看出,当每组的块容量为 8 块或 8 块以上时,所要的触发器个数及与门输入端个数很多,硬件实现的成本很高。这时,要采用分级的办法来实现。一般来说,所分级数越多,能节省的器件也越多,同时,器件的延迟时间也就越大。因此,采用多级方法实现的实质是用降低速度来换取节省器件。

例如,IBM 3033 机的 Cache 每组的块容量为 16,分 3 级来实现。从第 1 级到第 3 级分别为 4、2、2,它们的乘积为 16 块。因此,第 1 级要用 6 个触发器,第 2 级要用 $1 \times 4 = 4$ 个触

发器,第3级要用 $1 \times 2 \times 4 = 8$ 个触发器,总共用了 18 个触发器。如果不分级的话,从表 3.8 中看出需要触发器 120 个。采用分级方法节省了 100 多个触发器。

IBM 370/168 机的 Cache,采用组相联映象和变换方式。每组的块容量为 8,分成 2 级。第 1 级为 4,第 2 级为 2。总共需要触发器个数为: $6 + 1 \times 4 = 10$ 个。如果不分级,则总共需要 28 个触发器。

由于比较对法采用的是 LFU 算法,因此它的块失效率比较低。另外,它的工作速度比较高,只要使用很简单的组合逻辑就能找出最久没有被访问过的块。它的主要缺点是硬件实现相对比较复杂,需要比较多的触发器,特别是当每组的块容量比较大时。

3.3.3.4 堆栈法

在堆栈法中,用栈顶至栈底的先后次序来记录 Cache 同一组内的各个块被访问的先后次序。栈顶是最近被访问过的块,栈底是最久没有被访问过的块。当要替换时,从栈顶压入新的块,很自然,最久没有被访问过的块就从栈的底部被挤出堆栈。

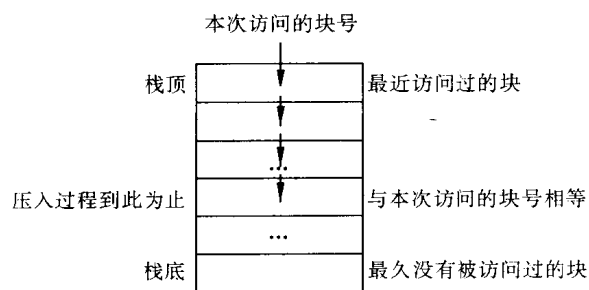


图 3.52 堆栈法的工作原理

这里所采用的堆栈法,与我们平时理解的先进先出(FIFO)或先进后出(FILO)堆栈有很大区别。为了在 Cache 中实现 LFU 替换算法,必须有一套新的管理规则。堆栈法的管理规则如下:

1. 把本次访问的块号与堆栈中保存的所有块号进行相联比较。如果发现有相等的,则 Cache 命中。这时,把本次访问的块号从栈顶压入,堆栈内各单元中的块号依次往下移,直至与本次访问的块号相等的那个单元为止,再往下的单元直至栈底都不变。

2. 如果相联比较没有发现相等的,则 Cache 块失效。这时,本次访问的块号从栈顶压入,堆栈内各单元的块号依次往下移,直至栈底,栈底单元中的块号被移出堆栈,它就是要被替换的块号。

因此,在堆栈法中,栈底永远保存着最久没有被访问过的块号,也就是下次要被替换出 Cache 的块号。

图 3.53 是一个用硬件实现堆栈法的逻辑图。图中,Cache 采用组相联映象及变换方式,每组的块容量为 4,因此,堆栈有 4 个存储单元,每个单元 2 位。堆栈用 D 型触发器实现,另外还要三个与门。图中, I_0I_1 是本次访问 Cache 的块号, A_0A_1 、 B_0B_1 、 C_0C_1 、 D_0D_1 是 4 个堆栈存储单元,它们分别存放最近访问 Cache 的 4 个块号,而且,在 A_0A_1 中存放的是

最近一次访问 Cache 的块号,然后依次存放。在 D_0D_1 中存放的是最久没有被访问过的块号,它的输出信号 D_0D_1 中就是下次将要被替换的 Cache 块号。CP 是一个时钟信号,当访问 Cache 时送来。 NA 、 NB 、 NC 是三个控制信号, NA 是指不是 A_0A_1 的意思,即本次访问 Cache 的块号 I_0I_1 与 A_0A_1 不相等, NA 、 NB 、 NC 的三个逻辑表达式如下:

$$NA = (I_0 \wedge \overline{A_0}) \vee (\overline{I_1} \wedge A_1)$$

$$NB = (I_0 \wedge \overline{B_0}) \vee (\overline{I_1} \wedge B_1)$$

$$NC = (I_0 \wedge \overline{C_0}) \vee (\overline{I_1} \wedge C_1)$$

在 NA 、 NB 、 NC 这三个信号的控制下,就能够重新排列保存在堆栈中的 Cache 块号访问顺序,从上到下按照访问 Cache 由近至远的顺序排列。

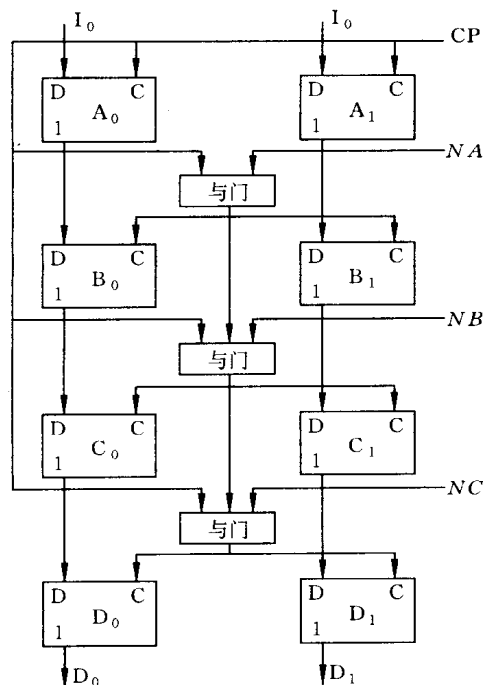


图 3.53 每组 4 块的堆栈法实现

堆栈法的主要优点有两个。一是它的块失效率比较低,因为它采用了 LFU 算法。二是它的硬件实现相对比较简单,除了必须的寄存器之外,其他控制逻辑很简单。它的主要缺点是速度比较低,这是因为它需要进行相联比较。因此,当每一组的块容量比较大时,不宜采用堆栈法。

堆栈与比较对法所用触发器的比例关系是:

$$\frac{G_b \cdot (G_b - 1)}{2} : G_b \cdot \log_2 G_b$$

其中, G_b 是 Cache 每一组的块容量。在 G_b 比较小时,两者的差别不大,当 G_b 大于 8 时,堆栈法所用的器件明显少于比较对法。

综上所述,对于 Cache 的替换算法,主要解决好以下三个问题:

1. 记录每次访问 Cache 的块号。可以用寄存器,也可以用计数器,或者用其他方法。总之,在实现 Cache 替换算法时,必须有时序逻辑。

2. 管理好所记录的 Cache 块号。例如,排序、计数、清除、置位等。其目的是为找出被替换的块号提供方便。

3. 根据记录和管理的结果,采用时序逻辑判断哪个块号是将被替换出去的块号。轮换法是要找出最早访问的块号,而另外三种方法都要找出最久没有被访问过的块号。

3.3.4 Cache 的性能分析

在计算机系统中设置 Cache 的一个主要目的是为了提高存储系统的速度,因此,人们最关心的一个问题就是 Cache 系统的加速比。另外,Cache 系统的一致性问题也很重要,这是由于 Cache 中保存的只是主存的一个副本,那么,这个副本与主存中的内容是否能保持一致,这是 Cache 能否可靠工作的一个关键问题。

3.3.4.1 Cache 系统的加速比

假设 Cache 的访问周期为 T_c ,主存储器的访问周期为 T_m ,Cache 系统的等效访问周期为 T ,Cache 的命中率为 H 。在本章开始曾经给出如下关系:

$$T = H \cdot T_c + (1 - H) \cdot T_m \quad (3.15)$$

则 Cache 系统的加速比 S_P 可以定义为:

$$S_P = \frac{T_m}{T} \quad (3.16)$$

加速比越高,说明 Cache 系统的等效访问速度与 Cache 的速度越接近,这正是大家所希望的。把(3.15)代入(3.16)中,得到:

$$S_P = \frac{T_m}{H \cdot T_c + (1 - H) \cdot T_m} = \frac{1}{(1 - H) + H \cdot \frac{T_c}{T_m}} \quad (3.17)$$

函数关系为:

$$S_P = f\left(H, \frac{T_m}{T_c}\right) \quad (3.18)$$

从这个关系式看到,Cache 系统的加速比 S_P 是命中率 H 和主存周期 T_m 与 Cache 周期 T_c 比值的函数。在 Cache 系统中,主存储器的访问周期 T_m 和 Cache 的访问周期 T_c 由于受所用器件的限制通常是确定的。因此,提高 Cache 系统的加速比 S_P 的最好途径是提高命中率 H 。

从(3.17)的关系中可以看出,加速比 S_P 与 Cache 命中率的关系是:当 $H=0.5$ 时,加速比 S_P 的期望值为 2。当 $H=0.75$ 时,加速比 S_P 的期望值为 4。当 $H=0.9$ 时,加速比 S_P 的期望值达到 10。当

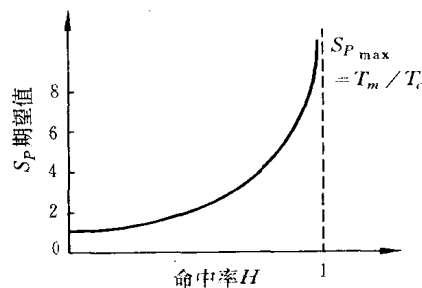


图 3.54 Cache 的加速比 S_P 与命中率 H 的关系

$H=1$ 时,加速比 S_p 达到最大值 $\frac{T_m}{T_c}$ 。

加速比 S_p 与 Cache 命中率 H 的关系如图 3.54 所示。由于命中率 H 的值一般都大于 0.9,能达到 0.99 以上,因此,从上面的关系中可以看出,实际上 Cache 的加速比 S_p 能够接近于它的最大值 $\frac{T_m}{T_c}$ 。

Cache 的命中率 H 主要与如下几个因素有关:程序在执行过程中的地址流分布情况;当发生 Cache 块失效时,所采用的替换算法;Cache 的容量;在组相联映象方式中,块的大小和分组的数目;所采用的 Cache 预取算法等。其中,地址流的分布情况是由程序本身决定的,系统设计人员一般无能为力。块替换算法已经在上一节中已经介绍过。Cache 预取算法将在以后专门介绍。以下,对影响 Cache 命中率的另外几个因素作简单的分析。

1. Cache 命中率与容量的关系

Cache 的命中率随它的容量的增加而提高,它们之间的关系曲线如图 3.55 所示。在 Cache 容量比较小的时候,命中率提高得非常快,随着 Cache 容量的增加,命中率提高的速度逐渐降低。当 Cache 容量增加到无穷大时,命中率可望达到 100%,但是,这实际上是做不到的。

在一般情况下,图 3.55 中的关系曲线可以近似地表示为 $H=1-S^{-0.5}$ 。因此,当 Cache 的容量达到一定值之后,再增加 Cache 容量,命中率的提高很少。

2. Cache 命中率与块大小的关系

在采用组相联和位选择组相联映象方式的 Cache 中,当 Cache 的容量一定时,块的大小对命中率的影响非常敏感。图 3.56 表示随着 Cache 块的由小到大的变化,命中率上升和下降的情况。

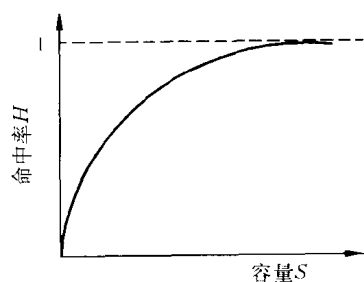


图 3.55 Cache 命中率 H 与它的容量 S 的关系

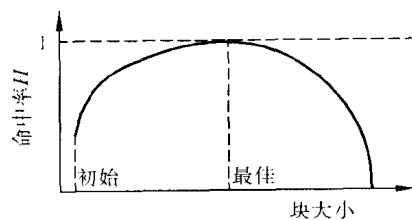


图 3.56 Cache 命中率与块大小的关系

开始时,块大小很小,例如只有一个存储字,这时的命中率 H 很低。随着块大小的增加,由于程序的空间局部性的作用,同一块中数据的利用率比较高,因此,Cache 的命中率增加。这种增加趋势在某一个最佳块大小处达到最大值。此后,命中率随着块大小的增加反而减小。

实际上,当块大小非常大时,进入 Cache 中的许多数据可能根本用不上。而且,随着块大小的增加,程序时间局部性的作用就会逐渐减弱。最后,当块大小等于整个 Cache 的容

量时,命中率将趋近于零。

3. Cache 命中率与组数的关系

当 Cache 的容量一定时,在采用组相联映象和变换方式的 Cache 中,分组的数目对于 Cache 命中率的影响是很明显的。随着组数的增加,Cache 的命中率要降低。当组数不太大时,例如,512 组以下,命中率的降低相当少,当组数超过一定数量时,命中率下降得非常快。

由于在组相联映象方式中,组间是采用直接映象方式的,只有组内采用全相联映象方式。当分组的数目增加时,主存中的某一块可以映象到 Cache 中的块数就将减少,从而导致命中率下降。

3.3.4.2 Cache 的一致性问题

在正常情况下,Cache 中所存放的内容应该是主存的部分副本。然而,由于以下两个原因,在一段时间内,主存某单元中的内容与 Cache 对应单元中的内容可能是不相同的。这样,就造成 Cache 与主存的不一致问题。

1. 如图 3.57(a)所示。由于 CPU 写 Cache,把 Cache 某单元中的内容从 X 修改成了 X',而主存对应单元中的内容还是 X,没有改变。

2. 如图 3.57(b)所示。由于从输入输出处理机或输入输出设备读入数据到主存储器,修改了主存某单元中的内容,把 X 修改成了 X',而 Cache 对应单元中的内容还是 X,没有改变。

当第 1 种情况发生时,如果把包括 X 在内的主存中的数据输出到设备去,或者在第 2 种情况发生时,CPU 读了 Cache 中的数据 X,都可能造成错误。

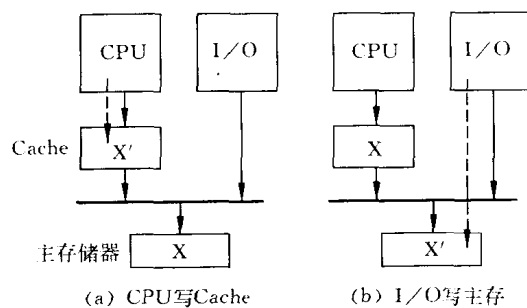


图 3.57 Cache 与主存不一致的两种情况

解决 Cache 与主存的不一致性问题,首先要选择合适的 Cache 更新算法。一般有两种 Cache 更新算法,写直达法和写回法,分别缩写为 WT(Write-through)和 WB(Write-back)。写直达法又称为写通过法,写回法又称为抵触修改法。

写回法是指 CPU 在执行写操作时,被写数据只写入 Cache,不写入主存。仅当需要替换时,才把已经修改过的 Cache 块写回到主存。在采用这种更新算法的 Cache 块表中,一般有一个“修改位”。当一块中的任何一个单元被修改时,这一块的修改位就被置“1”,否则这一块的修改位仍保持“0”。在需要替换这一块时,如果对应的修改位为“1”,则必须先把

这一块写回到主存中去之后,才能再调入新的块。如果修改位为“0”,则这一块不必写回主存,只要用新调入的块覆盖掉这一块即可。

写直达法是指 CPU 在执行写操作时,必须把数据同时写入 Cache 和主存。这样,在 Cache 的块表中就不需要“修改位”。当某一块需要替换时,也不必把这一块写回到主存中去,新调入的块可以立即把这一块覆盖掉。

比较写回法与写直达法的优缺点如下:

1. 可靠性:写直达法要优于写回法。这是因为写直达法能够始终保持 Cache 是主存的正确副本。如果 Cache 发生错误,可以从主存得到纠正。而写回法当发生图 3.57 所示的两种情况时,在一段时间内,Cache 并不是主存的正确副本。

2. 与主存的通信量:一般情况下,写回法少于写直达法。原因可以从两方面来分析。一方面,由于 Cache 的命中率一般很高,对于写回法,CPU 的绝大多数写操作只需写 Cache,不必写主存。另一方面,当写 Cache 发生块失效时,可能要写一个块到主存,而写直达法每次只写一个字到主存。而且,即使是读操作,当 Cache 不命中时,写回法也可能因为发生块替换而要写一块到主存。

总的来看,写直达法是把写主存的开销花在每次写 Cache 时再增加一个比写 Cache 要长得多的写主存操作上。而写回法是把写主存的开销集中在当发生 Cache 块失效时,可能要一次性地写一个块到主存。到底哪一种方法与主存的通信量更少,可以看下面的一个简单例子。

据统计,在访问存储器的操作中,写操作一般要占 10%到 34%。这里假设为 20%。Cache 的命中率为 99%。每块为 4 个字,主存的字长为一个字。当 Cache 发生块替换时,有 30%的块需要写回到主存,其余的块因为没有被修改过而不必写回主存。对于写直达法,写主存次数占总访存次数的 20%。而对于写回法,写主存次数占总访存次数的比例可以这样计算: $(1-99\%) \times 30\% \times 4 = 1.2\%$ 。因此,与主存的通信量,写回法仅是写直达法的十几分之一。

另外,无论是写回法,还是写直达法,当出现写 Cache 不命中时,都有一个是否要把包括所写字在内的一个块从主存读入 Cache 的问题。一般有两种方法。一种是“不按写分配法”,这种方法在发生写 Cache 不命中时,只把所要写的字写入主存,而包括所写字在内的一个块不从主存读入 Cache。另一种是“按写分配法”,这种方法在写 Cache 不命中时,除了完成把所写字写入主存之外,还要把包括所写字在内的一个块从主存读入 Cache。这两种方法的效果对于写回法和写直达法有所不同,但差别不大。目前,一般在写回法中采用按写分配法,而在写直达法中采用不按写分配法。

3. 控制的复杂性:写直达法比写回法简单。写回法要在块表中为每一块设置一个修改位,而且要对修改位进行管理和判断。而写直达法不需要设置修改位。另外,由于写直达法能够始终保持 Cache 是主存的正确副本,一旦 Cache 发生错误,可以从主存得到纠正。因此,Cache 通常只需要采用简单的奇偶校验即可。而对于写回法,Cache 中的块必须自己独立保证正确性,因此要采用相对比较复杂的纠错码。

4. 硬件实现的代价:写回法要比写直达法好。在写直达法中,因为每次写操作都要写主存,因此为了节省写主存所花费的时间,通常要采用一个高速小容量的缓冲存储器。

把要写主存的数据和地址先写到这个缓冲存储器中。在每次读主存时,也要首先判断所读的数据是否在这个缓冲存储器中。而写回法的硬件实现代价相对较低。

在多处理机(包括输入输出处理机或输入输出设备等)中,要解决 Cache 与主存的不一致性通常有如下几种方法:

1. 共享 Cache 法。造成 Cache 与主存不一致的根源是各个处理机(包括输入输出处理机)有各自的局部 Cache。当某一个处理机写了属于自己的局部 Cache 之后,造成 Cache 与主存不一致。因此,不允许各个处理机有属于自己的局部 Cache,即采用共享 Cache 能够从根本上解决 Cache 与主存的不一致问题。但是,共享 Cache 存在两个很难解决的问题。一个是当多个处理机都要访问同一个 Cache 时,Cache 的速度必然成为一个瓶颈。第二个是多端口 Cache 的仲裁既费设备又费时间。另外,目前的许多处理机都把一级 Cache 放在 CPU 内部,这样就根本不能实现共享。

共享 Cache 法,目前主要用于 CPU 与输入输出处理机共享同一个 Cache 的结构中。由于输入输出处理机访问存储器的频带宽度很低,一般不超过 Cache 频带宽度的 5%,因此,对 CPU 访问 Cache 不会造成太大影响。例如,Amdahl 470 系统就采用这种结构。

2. 作废法。各个处理机有属于自己的局部 Cache,也允许有的处理机(如输入输出处理机)没有属于自己的局部 Cache,而直接连接到主存储器上。当某一个处理机写属于自己的局部 Cache 时,同时要采用写直达法写主存,以保持属于自己的局部 Cache 与主存的一致性。或者当没有局部 Cache 的处理机写了主存之后,从而造成属于其他处理机的局部 Cache 与主存不一致,这时要发出一个特殊的命令,作废其他处理机的 Cache。可以作废全部 Cache 中的内容,也可以仅作废相关 Cache 中的相应块。前一种方法造成的损失很大,后一种方法实现起来比较复杂。

作废法可以全部用硬件实现,也可以用软件实现。用软件实现时,由操作系统经专门指令来作废 Cache。采用软件实现有一个很大的缺点,即使得 Cache 对系统程序员变成不透明的了。全部用硬件实现能够保持 Cache 对系统程序员的透明性,但硬件的代价比较高,控制也比较复杂。

3. 播写法。这种方法也允许各个处理机有属于自己的局部 Cache。当某一个处理机写属于自己的局部 Cache 时,采用广播的方法,把所写字的地址和内容送到公共的总线上。而其他有局部 Cache 的处理机都要随时监听公共总线上的地址,一旦发现总线上有与自己的局部 Cache 地址相符的信息,就把它从总线上拷贝下来,写到自己的局部 Cache 中。当然,同时也要采用写直达法把写 Cache 的一个字写回到主存中,以保持局部 Cache 与主存的一致性。

4. 目录表法。各个处理机也有属于自己的局部 Cache,并且要为各个 Cache 和主存建立目录表。在目录表中保存保证 Cache 一致性所需要的全部信息。目录表通常采用相联方式访问。当某一个处理机写属于自己的局部 Cache 时,同时要读目录表,判断这一块是否在其他处理机的局部 Cache 中。

目录表法,可以有中心目录表和分布目录表两种。中心目录表是用一个中心目录存放所有 Cache 目录的拷贝,即把保证所有 Cache 一致性的全部信息都保存在一个中心目录表中。中心目录表的优点是管理和维护比较方便,但是存在访问冲突和检索时间长两个缺

点。分布目录表是每个存储器(包括 Cache 和主存储器等)都维护各自的目录表。在目录表中记录各个存储器的本地信息和在哪个 Cache 中有该块的拷贝等信息。

5. 共享数据不存放在 Cache 中。这种方法只允许把指令和仅在本处理机中使用的数据存放在局部 Cache 中。共享数据,特别是需要改写的共享数据,如进程队列、信号灯、存储区的保护信息等,只能存放在共享的主存储器或共享的 Cache 中。

编译器在编译一个源程序时,必须把数据标记成可以存放在局部 Cache 中和不能存放在局部 Cache 中两种。而且,还必须要有专门的硬件来识别这些标记。这样,Cache 实际上对从事编译器设计的系统程序员是不透明的。如果某些标记还需要由程序设计人员来做的话,那么,Cache 对应用程序员也是不透明的了。

目前,很多单处理机都采用写回法,主要目的是为了减少 Cache 与主存之间的通信量。也有不少单处理机采用写直达法,原因是硬件的控制比较简单。而几乎所有的多处理机都采用写直达法,如 IBM 3033、IBM 370/168、VAX-11/780、Honeywell 66/60、Honeywell 66/80 等。Amdahl 的所有机器,IBM 3081 等机器采用写回法。

3.3.4.3 Cache 的预取算法

前面曾经提到,在一般情况下,预取能够大幅度提高 Cache 的命中率。具体地说,预取算法有如下几种:

1. 按需取。在一般 Cache 中采用的基本预取方法是按需取。即在出现 Cache 不命中时,把包括所要访问的字在内的一个块取到 Cache 中来。由于程序的局部性原理,只要 Cache 有适当的容量,块的大小选择得合适,就能获得一定的 Cache 命中率。

2. 恒预取。当 CPU 访问存储器时,无论 Cache 是否命中,都把紧接着访问字所在块的下一个块从主存取到 Cache 中。

3. 不命中预取。当 CPU 访问存储器时,如果 Cache 不命中,在把包括访问字在内的一块取到 Cache 中之后,还要把紧接着的下一块也取到 Cache 中。

应当注意,采用预取方法并不是在任何情况下都能提高命中率的,命中率与许多因素有关,例如,块大小(块容量)的影响。如果块容量很小,则预取的效果不大。如果块容量过大,一方面可能取进了许多没有用的数据,另一方面,由于块容量很大,Cache 的块数必然减少。根据前面的结论,由于块数减少,Cache 的命中率会降低。

由于预取必然增加 Cache 与主存之间的通信量,其中还要包括把被替换块写到主存中的通信量,因此,在考虑预取算法的效果时,不仅要看到命中率的提高,也要看到 Cache 与主存之间通信量的增加,要综合起来进行考虑。

从模拟实验的结果看,采用恒预取能使 Cache 的不命中率降低 75%~85%,而采用不命中预取能使 Cache 的不命中率降低 30%~40%。但是,恒预取所增加的 Cache 与主存之间的通信量要比不命中预取大很多。

3.4 三级存储系统

目前的大部分计算机系统中,既有虚拟存储器,也有 Cache。但是,在程序员看来,不

管存储系统的组织多么复杂,只看到一个存储器,而且也只关心一个存储器。这个存储器采用与主存储器完全相同的按地址随机访问的方式工作,它的等效访问速度接近于 Cache 的访问周期,等效存储容量是虚拟地址空间的容量。

按照本章第一节中给出的关于存储系统的定义,Cache、主存、磁盘这三个存储器可以分别构成“Cache-主存”和“主存-磁盘”两个存储系统,当然也可以构成一个“Cache-主存-磁盘”存储系统。具体地说,可以有如下几种做法:

1. 两个存储系统组织方式。就像上面两节中分别介绍的那样,有“Cache-主存”和“主存-磁盘”两个独立的存储系统。这种结构在有些资料上也称为物理地址 Cache,图3.58是这种组织方式的结构框图。

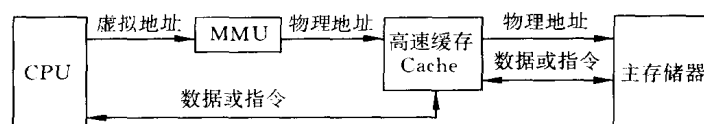


图 3.58 物理地址 Cache 存储系统

当 CPU 要访问存储器时,给出一个虚拟地址。由存储管理部件(MMU: memory management unit)中的地址变换部件把 CPU 发出的虚拟地址变换成主存物理地址。然后用主存物理地址访问 Cache。如果要访问的数据或指令在 Cache 被找到,则 Cache 命中,否则,发出 Cache 块失效,用这个物理地址访问主存储器,取出一块数据或指令装入 Cache,同时,也把 CPU 所需要的数据或指令送往 CPU。如 Intel 公司的 i486 和 DEC 公司的 VAX 8600 等处理机均采用两级存储系统。

2. 一个存储系统组织方式。把 Cache、主存和磁盘三个存储器组织在一起构成一个“Cache-主存-磁盘”存储系统。有些资料上把这种组织方式称为虚拟地址 Cache。结构框图如图 3.59 所示。

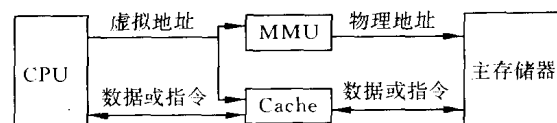


图 3.59 虚拟地址 Cache 存储系统

当 CPU 要访问存储器时,把虚拟地址直接送往存储管理部件 MMU 和 Cache。Cache 能够直接接受虚拟地址的访问,把 CPU 所需要的数据或指令找出来。如果 Cache 发生块失效,则用经过 MMU 变换得到的主存物理地址访问主存储器,把读出的一块数据或指令装入到 Cache 中。同时,也把 CPU 所需要的数据或指令送入 CPU。如 Intel 公司的 i860 等处理机采用这种组织方式。

3. 全 Cache 系统。这是最近才出现的一种新的存储器组织方式。在这种组织方式中,没有主存储器,只用 Cache 和磁盘(实际上只是磁盘中的一部分)两个存储器构成“Cache-磁盘”存储系统。

3.4.1 虚拟地址 Cache

在既有 Cache, 又有虚拟存储器的处理机中, 如果对 Cache 的访问仍采用主存实地址, 就要把虚拟地址首先变换成主存实地址, 然后才能访问 Cache, 这样必然增加访问 Cache 所花费的时间, 至少要增加一个查主存快表的时间。因此, 在许多系统中, 采取直接用虚拟地址访问 Cache 方法, 如图 3.60 所示。

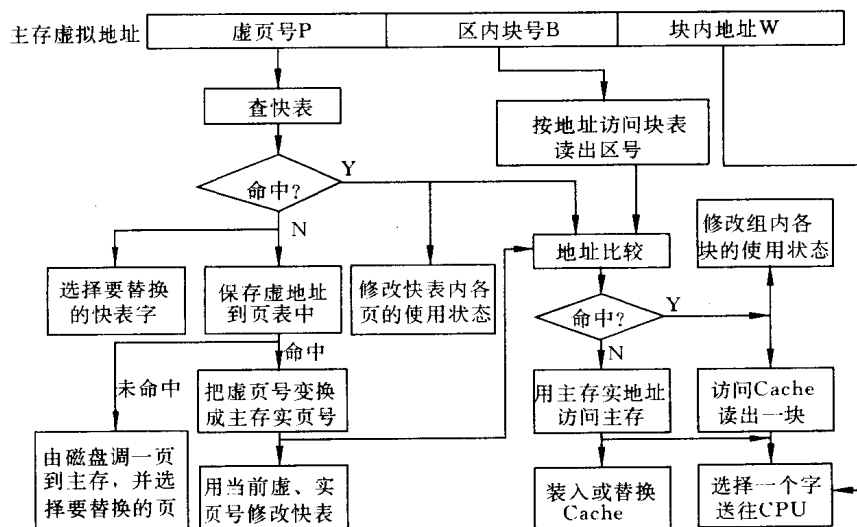


图 3.60 一种虚拟地址 Cache 的地址变换过程

在图 3.60 中, 虚拟存储器采用位选择组相联映象和地址变换方式。为了加快地址变换的速度, 使虚拟存储器中的一页恰好就是主存储器的一个区, 可以直接用虚拟地址中的区内块号 B 按地址访问 Cache 的块表, 从块表中读出主存的区号 E 和对应的 Cache 块号 b, 这个区号实际上也就是页号 P。在访问 Cache 块表的同时, 用虚拟地址中的虚页号访问快表。

如果快表命中, 就用从块表中读出来的主存区号 E 与从快表中得到的主存实页号 P 进行相等比较。若比较结果相等, 则 Cache 命中。这时, 把虚拟地址中的区内块号 B 直接作为 Cache 地址中的组号 g, 并从块表相应单元中读出 Cache 的组内块号 b, 把虚拟地址中的块内地址 W 直接作为 Cache 地址中的块内地址 w。用 g、b、w 三部分拼接就得到的 Cache 地址, 用这个地址访问 Cache, 读出一个字送往 CPU。若 Cache 不命中, 则用主存实页号 P 拼接上虚拟地址中的区内块号 B 和块内地址 W, 得到主存实地址去访问主存储器, 把读出的一个字送往 CPU。同时把包括这个字在内的一块都从主存储器中读出来装入到 Cache 中去。如果, 这时 Cache 已经满, 还要采用某种 Cache 替换算法先把不常用的一块替换到主存中去。

如果快表没有命中, 要通过软件去查存放在主存中的慢表。以下的工作过程与页式虚拟存储器或段页式虚拟存储器相同。

3.4.2 全 Cache 技术

以上介绍了两个存储系统,一个是由 Cache 和主存储器构成的 Cache 存储系统,另一个是由主存储器和辅助存储器(磁盘存储器中的一部分)构成的虚拟存储器。但是,从系统程序员所看到的存储系统,或者从 CPU 对存储系统的要求看,只看到一个存储器,而且也希望只有一个存储器,就是前面提到过的存储系统的等效存储器。这个存储器工作速度很高,采用按地址随机访问的方式工作。至于这个存储器如何构成,软件与硬件的分工等,则是具体的实现方法而已。

最简单,也是最直接的实现方法是:用一个速度很高(与 Cache 的速度相当)、存储容量很大(与虚拟地址空间的容量相当)、能够按地址随机访问的存储器来实现。当然,还要考虑实现的成本能否被大家所接受。但是,就目前的硬件技术来看,采用这种方法来实现是根本做不到的。也许在将来能够做到。如果真有这一天的话,那么,本章中所讲述的几乎全部内容都将是没有用的。

根据目前的硬件实现技术,除了上面介绍的采用 Cache、主存和辅存(磁盘存储器中的一部分)三个存储器构成两个两级存储系统和一个三级存储系统方法之外,还有一种新的实现方法。就是不用主存储器,只用 Cache 和辅存两个存储器构成“Cache-辅存”存储系统。这种存储系统简称为全 Cache(all-Cache)存储系统。

全 Cache 存储系统的等效访问周期与 Cache 很接近,等效存储容量就是虚拟地址空间的容量。

目前,全 Cache 存储系统还处于实验阶段,并没有商用化。在多处理机系统中,一种全 Cache 存储系统的方案如图 3.61 所示。

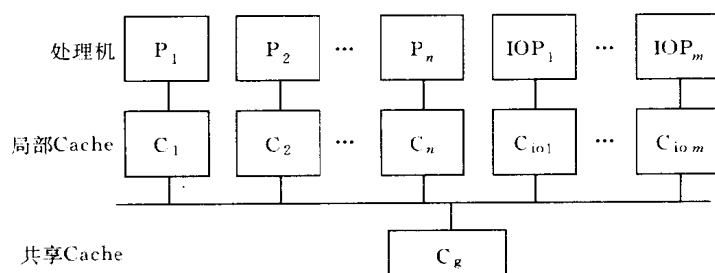


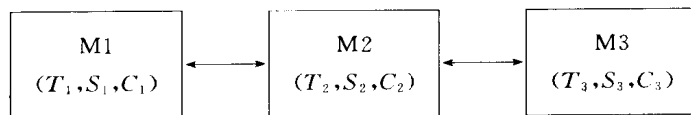
图 3.61 多处理机系统中的全 Cache 存储系统

由于作为辅存的磁盘存储器的基本访问单位是物理块,每个物理块的容量是 512 个字节,因此,与磁盘存储器连接的局部 Cache 的块容量一般也是 512 个字节。其他 Cache 的块容量可以比 512 个字节小,也可以比 512 个字节大,一般只要取整数倍的关系即可。

最后,应当指出,上面介绍的虚拟存储系统和 Cache 存储系统只是在目前的硬件和软件实现技术条件下提出来的。随着硬件和软件技术的发展,这些实现技术也并不总是有用的。例如,磁盘阵列和闪烁存储器的逐渐实用化,对实现全 Cache 存储系统是非常有力的支持。特别是闪烁存储器,很有可能成为代替磁盘存储器和主存储器的一种新型存储器,从而改变目前的存储系统组织方式。

习 题 三

- 3.1 什么是存储系统？对于一个由两个存储器 M_1 和 M_2 构成的存储系统，设 M_1 的命中率为 h ，两个存储器的存储容量分别为 s_1 和 s_2 ，访问速度分别为 t_1 和 t_2 ，每千字节的价格分别为 c_1 和 c_2 。
- (1) 在什么条件下，整个存储系统的每千字节平均价格会接近于 c_2 ？
 - (2) 写出这个存储系统的等效访问时间 t_a 的表达式。
 - (3) 假设存储系统访问效率 $e = \frac{t_1}{t_a}$ ，两个存储器的速度比 $r = \frac{t_2}{t_1}$ 。试以速度比 r 和命中率 h 来表示访问效率 e 。
 - (4) 分别画出 $r=5, 20$ 和 100 时，访问效率 e 和命中率 h 的关系图。
 - (5) 如果 $r=100$ ，为了使访问效率 $e > 0.95$ ，要求命中率 h 是多少？
 - (6) 对于(5)所要求的命中率实际上很难达到。假设实际的命中率只能达到 0.96 。现采用一种缓冲技术来解决这个问题。当访问 $M1$ 不命中时，把包括被访问数据在内的一个数据块都从 $M2$ 取到 $M1$ 中，并假设被取到 $M1$ 中的每个数据平均可以被重复访问 5 次。请设计缓冲深度（即每次从 $M2$ 取到 $M1$ 中的数据块的大小）。
- 3.2 由三个访问速度、存储容量和每位价格都不相同的存储器构成一个存储系统，其中， $M1$ 靠近 CPU。回答下列问题：



- (1) 写出这个三级存储系统的等效访问时间 T ，等效存储容量 S 和等效每位价格 C 的表达式。
 - (2) 在什么条件下，整个存储系统的每位平均价格接近于 C_3 ？
- 3.3 要求设计一个由 Cache 和主存构成的两级存储系统，已知 Cache 的容量有三种选择：64K 字节、128K 字节和 256K 字节，它们的命中率分别为 0.7 、 0.9 和 0.98 。主存的容量为 4M 字节。并设两个存储器的访问时间分别为 t_1 和 t_2 ，每字节的价格分别为 c_1 和 c_2 。如果 $c_1 = 20c_2$ ，为 $t_2 = 10t_1$ 。
- (1) 在 $t_1 = 20\text{ns}$ 的条件下，分别计算三种 Cache 的等效访问时间。
 - (2) 如果 $c_2 = 0.2$ 美元/K 字节，分别计算三种 Cache 每字节的平均价格。
 - (3) 根据三种 Cache 的等效访问时间和每字节的平均价格排列次序。
 - (4) 根据等效访问时间和平均价格的乘积，选择最优的设计。
- 3.4 对于一个由 Cache 和主存构成的两级存储系统，已知 Cache 的容量 $s_1 = 512$ 字节，主存的容量 s_2 未知；Cache 的命中率为 $h = 0.95$ ；Cache 的访问时间 $t_1 = 20\text{ns}$ ，主存的访问时间 t_2 未知；Cache 的价格 $c_1 = 0.01$ 美元/字节，主存的价格 $c_2 = 0.5$ 美元/K 字节。要求两个存储器的总价格不能超过 15 000 美元。

- (1) 在不超过预算的范围内,可能得到的主存储器最大容量是多少?
- (2) 为了使这个两级存储系统的等效访问时间达到 40ns ,主存储器的访问时间 t_2 应该是多少?
- 3.5 假设程序中出现转移指令且转移成功的概率为 0.1 ,设计一个采用低位交叉方式访问的多体存储器,要求每增加一个存储体在一个存储周期中能够访问到的平均指令条数增加 0.2 条以上,请计算最多的并行存储体的个数。
- 3.6 一台处理机的运算速度为 1GIPS ,每执行一条指令平均需要取指令一条和读/写数据两个,输入输出系统对存储器的访问可以忽略不计。主存储器采用 DRAM 芯片,工作周期为 150ns ,请设计存储系统方案,可以采取哪些措施来匹配存储器与 CPU 之间的速度差距?每一种措施大概能够弥补多少倍数?
- 3.7 有 16 个存储器模块,每个模块的容量为 4M 字节,字长为 32 位。现在要用这 16 个存储器模块构成一个主存储器,有如下几种组织方式:
- 方式 1: 16 个存储器模块采用高位交叉方式构成存储器。
- 方式 2: 16 个存储器模块构成并行访问存储器。
- 方式 3: 16 个存储器模块采用低位交叉方式构成存储器。
- 方式 4: 2 路高位交叉 8 路低位交叉构成存储器。
- 方式 5: 4 路高位交叉 4 路低位交叉构成存储器。
- 方式 6: 4 路并行访问 4 路低位交叉构成存储器。
- (1) 写出各种存储器的地址格式。
- (2) 比较各种存储器的优缺点。
- (3) 不考虑访问冲突,计算各种存储器的频带宽度。
- (4) 画出各种存储器的逻辑示意图。
- 3.8 一个 16×16 的矩阵,要求在一个存储器周期内实现按行、按列、按对角线和按反对角线的无冲突访问。至少需要多少个存储体?写出矩阵的各元素在各个存储体中存放的位置。
- 3.9 一个页式虚拟存储器的虚存空间大小为 4GB ,页面大小为 4KB ,每个页表存储字要占用 4 个字节。
- (1) 计算这个页式虚拟存储器需要采用几级页表?
- (2) 如果要求页表所占总的主存页面数最小,请分配每一级页表的实际存储容量各为多少字节?
- (3) 页表的哪些部分必须存放在主存中?哪些可以放在辅存中?
- 3.10 在一个采用快、慢表的虚拟存储器中,访问主存的命中率为 99.99% ,访问快表的命中率为 99% ,经快表进行地址变换所需要的时间仅是经过查页表进行地址变换时间的十分之一,磁盘存储器的访问速度仅是主存储器速度的万分之一。
- (1) 计算这个虚拟存储器的等效访问时间。
- (2) 计算这个虚拟存储器的访问效率。
- 3.11 一个页式虚拟存储器按字节编址,页面大小为 1K 字节,每个数据的字长为 4 个字节。现有一个程序的页表如下:

虚页号	装入标志	主存页号	修改标志	访问方式
0	1	2	0	RW
1	1	3	0	R
2	0	0	0	R
3	1	1	0	X
4	0	0	0	RW
5	1	0	0	R
6	0	0	0	X

表中的装入标志为“1”表示该虚页已经装入主存,为“0”则表示还未装入主存。修改标志为“0”表示该页还没有被修改过,为“1”则表示该页已经被修改过。访问方式“RW”表示该页可以读可以写,但不能作为指令来执行;“R”表示该页只能读,不能写和执行;“X”表示该页只能作为指令来执行,不能读和写。

虚地址经变址寻址和基址寻址(B)+(X)+D形成。现有一个程序,出现下列访问主存的操作:

序号	操 作	虚 存 地 址		
		(B)	(X)	D
1	取 数	124	30	50
2	取 数	2 000	1 000	60
3	存 数	4 000	2 000	600
4	存 数	1 200	4 600	60
5	取 数	3 000	640	100
6	取 数	4 096	500	20
7	加并存数	400	1 200	80
8	加并存数	36	360	64
9	转 移	2 500	600	100
10	转 移	3 600	1 200	56

(1) 列出产生主存页面失效的操作序号。

(2) 如果不发生主存页面失效的话,计算访问主存的物理地址。

(3) 列出被修改过的主存页面号。

(4) 列出非法操作的序号。

- 3.12 一个有快表和慢表的页式虚拟存储器,最多有 64 个用户,每个用户最多要用 1 024 个页面,每页 4K 字节,主存容量 8M 字节。

- (1) 写出多用户虚地址的格式,并标出各字段的长度。
 - (2) 写出主存地址的格式,并标出各字段的长度。
 - (3) 快表的字长为多少位?分几个字段?各字段的长度为多少位?
 - (4) 慢表的容量是多少个存储字?每个存储字的长度为多少位?
 - (5) 画出多用户虚地址经快表或慢表变换成主存实地址的逻辑示意图。
- 3.13 一个虚拟存储器按字节编址,最多有 256 个用户,每个用户最多要用 4 096 页,每页 1K 字节。主存容量 16M 字节,快表按地址访问,共 32 个存储字,快表地址码经散列变换得到,为减少散列冲突,快表分为两组,有两套独立的相等比较电路。
- (1) 写出多用户虚地址和主存地址的格式,并标出各字段的长度。
 - (2) 散列变换部件的输入位数和输出位数各为多少?
 - (3) 每个相等比较电路的位数是多少?
 - (4) 快表每个存储字的总长度为多少位?分哪几个字段?各字段的长度为多少位?
 - (5) 画出多用户虚地址经快表变换成主存地址的逻辑示意图。
- 3.14 在页式虚拟存储器中,一个程序由 P1~P5 共 5 个页面组成。在程序执行过程中依次访问到的页面如下:
P2,P3,P2,P1,P5,P2,P4,P5,P3,P2,P5,P2
- 假设系统分配给这个程序的主存有 3 个页面,分别采用 FIFO、LFU 和 OPT 三种页面替换算法对这 3 页主存进行调度。
- (1) 画出主存页面调入、替换和命中的情况表。
 - (2) 统计三种页面替换算法的页命中率。
- 3.15 一个程序由 5 个虚页组成,采用 LFU 替换算法,在程序执行过程中依次访问的页地址流如下:
P4,P5,P3,P2,P5,P1,P3,P2,P3,P5,P1,P3
- (1) 可能的最高页命中率是多少?
 - (2) 至少要分配给该程序多少个主存页面才能获得最高的命中率?
 - (3) 如果在程序执行过程中每访问一个页面,平均要对该页面内的存储单元访问 1 024 次,求访问存储单元的命中率。
- 3.16 一个程序由 1 200 条指令组成,每条指令的字长均为 4 个字节。假设这个程序访问虚拟存储器的字地址流依次为:12,40,260,280,180,800,500,560,600,1 100,1 200,1 000。采用 FIFO 页面替换算法,分配给这个程序的主存容量为 2 048 个字节。在下列不同的页面大小情况下,分别写出这个程序执行过程中依次访问的虚存页地址流,并计算主存的页命中率。
- (1) 页面大小为 1 024 个字节。
 - (2) 页面大小为 512 个字节。
 - (3) 页面大小为 2 048 个字节
 - (4) 比较(1)、(2)、(3),可以得出什么结论?
 - (5) 如果把分配给该程序的主存容量增加到 4 096 个字节,页面大小为 1 024 个字节,再做一遍,由此又可以得出什么结论?

- 3.17 在计算机系统中设置虚拟存储器和 Cache 的主要目的各是什么？试列举出这两种存储系统在具体实现时至少 4 个方面的差别，并说明主要理由。
- 3.18 试比较下列 5 种 Cache 的主要优缺点：
- 第 1 种：直接映象方式 Cache
 - 第 2 种：全相联映象方式 Cache
 - 第 3 种：组相联映象方式 Cache
 - 第 4 种：位选择组相联映象方式 Cache
 - 第 5 种：段相联映象方式 Cache
- 并回答下列问题：
- (1) 根据硬件的复杂性和实现所需要的成本排出 5 种 Cache 的次序，并说明理由。
 - (2) 根据块替换算法实现的灵活性排出 5 种 Cache 的次序，并说明理由。
 - (3) 解释各种 Cache 的块映象方式对命中率的影响。
 - (4) 在采用组相联映象方式的 Cache 中，说明块大小、组数、Cache 容量、块替换算法等对 Cache 性能的影响。
- 3.19 假设在一个采用组相联映象方式的 Cache 中，主存由 B0~B7 共 8 块组成，Cache 有 2 组，每组 2 块，每块的大小为 16 个字节，采用 LFU 块替换算法。在一个程序执行过程中依次访问这个 Cache 的块地址流如下：
- B6, B2, B4, B1, B4, B6, B3, B0, B4, B5, B7, B3
- (1) 写出主存地址的格式，并标出各字段的长度。
 - (2) 写出 Cache 地址的格式，并标出各字段的长度。
 - (3) 画出主存与 Cache 之间各个块的映象对应关系。
 - (4) 如果 Cache 的各个块号为 C0、C1、C2 和 C3，列出程序执行过程中 Cache 的块地址流情况。
 - (5) 如果采用 FIFO 替换算法，计算 Cache 的块命中率。
 - (6) 采用 LFU 替换算法，计算 Cache 的块命中率。
 - (7) 如果改为全相联映象方式，再做(5)和(6)，可以得出什么结论？
 - (8) 如果在程序执行过程中，每从主存装入一块到 Cache，则平均要对这个块访问 16 次。请计算在这种情况下 Cache 命中率。
- 3.20 在一个采用组相联映象方式的 Cache 中，Cache 的容量为 16KB。主存采用模 8 低位交叉方式访问，每个存储体的字长为 32 位，总容量为 8MB。要求 Cache 的每一块在一个主存周期内分别从 8 个存储体中取得，Cache 的每一组内共有 4 块。要求采用按地址访问存储器方式构成相联目录表，实现主存地址到 Cache 地址的变换，并采用 8 个相等比较电路。
- (1) 设计主存地址格式，并标出各字段的长度。
 - (2) 设计 Cache 地址格式，并标出各字段的长度。
 - (3) 相联目录表的行数(即地址个数)是多少？
 - (4) 设计相联目录表每一行的格式，并标出每一个字段的长度。
 - (5) 每个比较电路的位数是多少？

- (6) 画出主存地址经相联目录表变换成 Cache 地址的逻辑示意图
3. 21 一个采用位选择组相联映象方式的 Cache, 要求 Cache 的每一块在一个主存周期内取得。主存采用 4 个存储体的低位交叉方式访问, 每个存储体的字长为 4 个字节, 总容量为 1MB。Cache 的容量为 1KB, 每一组内有 4 块。采用按地址访问存储器构成相联目录表, 实现主存地址到 Cache 地址的变换, 采用 4 个相等比较电路。
- (1) 设计主存地址格式, 并标出各字段的长度。
 - (2) 设计 Cache 地址格式, 并标出各字段的长度。
 - (3) 设计相联目录表结构, 求出该表的行数及每一行的格式。
 - (4) 每一个比较电路的位数是多少?
 - (5) 画出实现位选择组相联地址变换的逻辑示意图。
3. 22 一个采用组相联映象方式的 Cache 共有 8 块, 分成两组, 用硬件的比较对法实现 LFU 块替换算法。
- (1) 共需要多少个触发器? 多少个与门?
 - (2) 画出其中一组的逻辑图。
3. 23 对于一个采用组相联映象方式和 FIFO 替换算法的 Cache, 发现它的等效访问时间太长, 为此, 提出如下改进建议:
- (1) 增大主存的容量。
 - (2) 提高主存的速度。
 - (3) 增大 Cache 的容量。
 - (4) 提高 Cache 的速度。
 - (5) Cache 的总容量和组大小不变, 增大块的大小。
 - (6) Cache 的总容量和块大小不变, 增大组的大小。
 - (7) Cache 的总容量和块大小不变, 增加组数。
 - (8) 替换算法由 FIFO 该为 LFU。
- 请分析以上改进建议对等效访问时间有何影响, 其影响的程度如何?
3. 24 在一个 Cache 中, 有 n 块要采用相联映象方式访问, 现设计一个 n 行 \times n 列的触发器阵列来实现 LRU 替换算法, 其行和列的编号都从 1 到 n 。当 Cache 的第 n 块被访问时, 先把触发器阵列的第 n 行全部置“1”, 然后把第 n 列全部置“0”。这样, 在任何时刻, 触发器阵列中二进制值最小的行号就是近期最少访问过的 Cache 块号。现假设 $n=4$, 访问 Cache 的块地址流为 1、2、3、4、4、2、3、1、4、3、4、3、2, 请验证这一结论的正确性。

第四章 输入输出系统

4.1 输入输出原理

处理机在运行过程中所需要的程序和数据要从外部输入,运算结果要输出到外部去。处理机通常还要与其它处理机发生关系,甚至处理机必须在人的操作下才能完成人们所希望做的工作。

输入输出系统能够提供处理机与外部世界进行交往或通信的各种手段。所谓外部世界是指处理机以外的需要与处理机交换信息的人和物,主要包括本地和远程用户、系统操作员、操作控制台、输入输出设备、辅助存储器、其他处理机、各种通信设备和虚拟现实系统等。由于人也是通过各种设备来访问处理机的,因此,把人以外的各种设备统称为输入输出设备,或外围设备。

在计算机系统中,通常把处理机和主存储器之外的部分统称为输入输出系统,它包括输入输出设备、输入输出接口和输入输出软件等。

4.1.1 输入输出系统的特点

输入输出系统品种繁多,性能迥异,使输入输出系统成为计算机系统中最具多样性和复杂性的部分。它的多样性不仅表现在输入输出设备的品种、功能、技术指标等诸方面,还表现在这些设备广泛地涉及到机、光、电、磁、声、自动控制等多种学科。从计算机学科来看,输入输出系统最典型地反映着硬件与软件的相互结合。

现代计算机输入输出系统的复杂性一般都隐藏在操作系统之中,一般用户只需通过一些简单的命令或程序调用就能使用各种输入输出设备,而无需了解各种输入输出设备的具体工作细节。

输入输出系统的复杂性还表现在处理机本身和操作系统所产生的一系列随机事件也要调用输入输出系统来进行处理,例如,中断和自陷等。

图 4.1 说明输入输出系统所处理的外部世界的多样性和复杂性。面对图中所列各类外围设备和随机事件的类型及它们要求处理机提供的处理环境,如果不借助功能强大和灵活多变的软件,输入输出系统的多样性、复杂性与使用的统一性、方便性之间的矛盾是不可能圆满解决的。

输入输出系统的特点集中反映在异步性、实时性和与设备无关性三个项基本要求上,它们对输入输出系统的组织产生决定性的影响。

4.1.1.1 异步性

外围设备相对于处理机通常是异步工作的。如果不考虑缓冲存储器的作用,终端设备

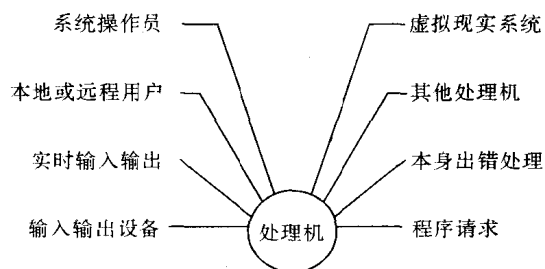


图 4.1 处理机与外部的联系

通常按照人操作键盘的速度输入字符到处理机中,同时把所输入的字符显示到显示器上。打印机通常按照它自身的速度,每分钟打印一定行数的字符数。输入输出设备的工作在很大程度上独立于处理机之外,通常不使用统一的中央时钟,各个设备按照自己的时钟工作,但又要在某些时刻接受处理机的控制。

外围设备的工作过程通常是这样的,当设备准备好与处理机交往时(对于输入设备是指数据寄存器满,对于输出设备是指数据寄存器空),要向处理机申请服务。对于处理机来说,这个时刻一般是随机的,两次申请之间可能要经过很长时间,这就造成了输入输出相对于处理机的异步性和时间上的任意性。

当一个处理机管理多台外围设备时,必须做到在任意两次处理机与设备交往的时刻之间,处理机仍然能够全速运行它本身的程序,或者管理其他外围设备,从而保证处理机与外围设备之间,外围设备与外围设备之间能够并行工作,无需互相等待。为了能够实现一个处理机与多台外围设备并行工作,必须采用中断输入输出方式或直接存储器存取(DMA)方式工作。

4.1.1.2 实时性

当外围设备与处理机交往时,由于设备的类型不同,它们的工作步调是很不相同的,信息传输的速率也相差悬殊,传送方式极不统一。有的设备每次只传送一个字符,如打印机和终端设备等,传输速度为每秒钟几个到几十个字符。有的设备按数据块或按文件为单位传送,如磁盘、磁带存储器等,每秒钟要传送几到几十兆字符。处理机必须按照不同设备所要求传送方式和传输速率不失时机地为设备提供服务,包括从设备接收数据,向设备发送数据及对设备进行控制等。如果错过了服务的时机,就可能丢失数据,或造成外围设备工作的错误。

用作实时控制的计算机系统(例如,工业过程控制,导弹、卫星等的控制)对时间性的要求非常强,如果处理机提供的服务不及时,很可能造成巨大的损失,甚至造成人身伤害。

对于处理机本身的硬件或软件错误,例如,电源故障、数据校验错、页面失效、非法指令、地址越界等,处理机也必须及时地给予处理。

处理机为了能够为各种不同类型的设备提供服务,必须具有与各种设备相配合的多种工作方式,包括程序控制方式、中断方式、直接存储器存取方式等。这就是输入输出设备的实时性要求。

4.1.1.3 与设备无关性

外围设备的类型、规格、特性多种多样,它们输出数据和接收数据的方式、数据的格式差异很大,因此,设备与处理机的连接方式也很不相同。计算机系统为了能够适应各种外围设备的不同要求,规定了一些独立于具体设备的标准接口。例如,串行接口、并行接口、SCSI(small computer system interface)接口等

各种外围设备必须根据自己的特点和要求,选择其中的一种标准接口与处理机进行连接。凡是连接到同一种标准接口上的不同类型的设备,它们之间的差异必须由设备本身的控制器通过硬件和软件来进行填补。这样,处理机本身就无需了解各种外围设备特定的具体工作细节,可以采用统一的硬件和软件对品种繁多的设备进行管理。计算机系统的使用者也只需通过操作系统提供的高级命令或程序请求来使用各种各样复杂的外围设备。在需要更换外围设备时,各种不同型号、不同生产厂家的设备都可以直接通过标准接口与计算机系统连接。

最近,在某些计算机系统中已经实现了所谓即插即用技术。例如,在 Windows 95 和 Windows NT 操作系统中,凡是经过 Microsoft 公司测试过的机型及外围设备都可以直接连接,由操作系统统一分配输入输出地址、中断号和 DMA 接口号,并采用操作系统中提供的驱动软件。在操作系统安装过程中能够自动识别各种外围设备,自动调用相应的驱动软件,无需用户自己安装。对于新加入的外围设备,操作系统也能自动识别,为它分配输入输出地址、中断号、DMA 接口号,并调用相应的驱动软件。

4.1.2 输入输出系统的组织方式

输入输出设备的异步性、实时性、与设备无关性三个特点是现代计算机系统必须具备的共同特性。其中,异步性体现了输入输出系统相对于处理机的独立自主关系;实时性反映了按照不同设备响应时间的不同要求,划分和实现输入输出系统内部不同功能之间的关系;与设备无关性贯彻了输入输出系统标准化接口与非标准外围设备之间的关系。根据各种外围设备的不同特点处理好这三方面的关系,就成为输入输出系统组织的基本内容。具体地说,针对异步性,采用自治控制的方法;针对实时性,采用层次结构的方法;针对与设备无关性,采用分类处理的方法。

4.1.2.1 自治控制

在一般计算机系统中,输入输出系统是一个独立于处理机之外的自治系统。外围设备本身通过它内部的控制器独立担负许多输入输出功能,只在某些必要的时刻才要求处理机给予很少的服务。

许多外围设备,只要在进入工作状态时,由处理机发出启动设备的命令,并组织好送给输入输出系统自己执行的控制程序,在结束工作时,对数据进行必要的处理,并发出停止设备工作的命令。具体的输入输出过程、数据的传送可以根本不要处理机来干预。对于

有些设备,也只需要少量的干预。

到底哪些功能由外围设备自己完成,哪些功能需要处理机来做,这种功能上的划分要随计算机系统的设计要求和不同的外围设备种类来具体确定。所以,自治控制实际上也就是把外围设备所要完成的功能分散开来,即把设备的输入输出功能最大限度地从处理机中分离出来,由专门的设备控制器通过它自身的硬件和软件去完成,从而使处理机能够摆脱繁重的输入输出任务。

4.1.2.2 层次结构

对于不同类型的外围设备,输入输出系统要完成的具体内容是不同的。采用可编程序来进行控制,能比较好地适应各种不同设备的控制需要,赋予固定的硬件设备以很大的灵活性。

在外围设备比较多的情况下,输入输出系统一般要采用层次结构来进行组织。靠近处理机和主存储器的最高层采用标准的控制功能,如输入输出处理机、输入输出通道等。中间层是标准接口。外围设备通过设备控制器与标准接口相连接。层次结构思想的框图如图 4.2 所示。

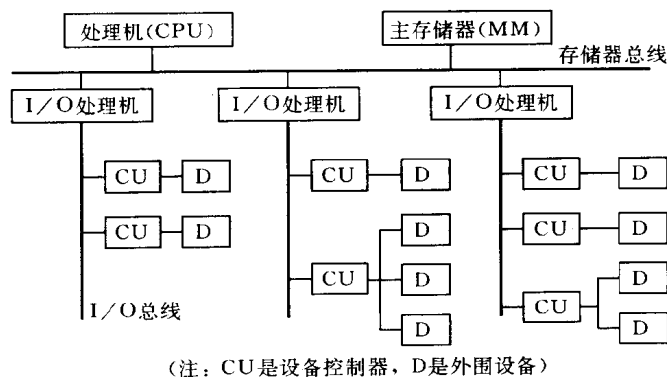


图 4.2 输入输出系统的层次结构

外围设备一般要按照工作方式和工作速度等进行分类,不同类型的设备连接到不同的输入输出通道或输入输出处理机上。输入输出通道或输入输出处理机通常是一台小型的专用处理机,它能够执行与输入输出操作有关的程序。当需要对某一外围设备实现输入或输出操作时,CPU 只需要组织好输入输出程序,并启动相关的输入输出通道或输入输出处理机即可。具体的输入输出操作在输入输出通道或输入输出处理机的控制下完成,不需要 CPU 干预。

输入输出通道或输入输出处理机执行 CPU 为它组织好的程序,通过输入输出指令控制标准接口,标准接口在指令控制下发出一系列标准的控制信号,送往设备控制器。设备控制器根据具体设备的不同需要,在自己的硬件和软件的控制下,可以产生设备所需要的各种非标准信号,最终完成各种输入输出操作。

4.1.2.3 分类组织

输入输出系统通常要根据各种外围设备的不同性质分类进行组织。可以根据工作方式、工作速度及使用场合等进行分类。

如果按照工作速度进行分类,可以分为面向字符的设备(character-oriented device)和面向数据块的设备(block-oriented device)两类。

面向字符的设备主要是指工作速度比较低的机电类设备,例如字符终端、打字机等。这类设备通常以字符或字作为与处理机之间传送数据的基本单位,在设备控制器中一般只设置能够存放一个或少数几个字符的缓冲寄存器。对于输入设备,缓冲寄存器用来作为 CPU 读取字符数据的源寄存器,而对于输出设备,缓冲寄存器用来作为 CPU 送来字符数据的目的寄存器。

面向字符的设备通常只在启动、停止或已经准备好要发送或接收数据的情况下才要求 CPU 作短时间的服务。而在主存储器中也只需要指定少数的存储单元作为数据的预处理区,以便对数据进行打包或校验等处理。

面向数据块的设备主要指工作速度比较高的外围设备,例如,磁盘、磁带、光盘等辅助存储器、行式打印机、卡片阅读机等。这类设备通常以一定数量的字符或字组成的数据块作为与处理机之间传送数据的基本单位。数据块的大小取决于数据在设备中的物理分布情况。在磁盘、磁带、光盘等辅助存储器中通常以 512 个字节作为一个数据块,在行式打印机、卡片阅读机等中通常以一行字符作为一个数据块。对于高速设备的数据块传送请求, CPU 必须及时响应,并给予处理,否则有可能丢失数据。在设备控制器中可以设置能够存放一个数据块的缓冲存储器,也可以只设置能够存放一个或少数几个字符的缓冲寄存器。但在主存储器中一般要开辟一个比较大的数据缓冲区,以便对数据进行前处理(输出数据时)或后处理(输入数据时)。

面向数据块的设备一般不需要 CPU 对每一个字符进行输入输出服务。这类设备一旦被启动,在设备控制器的控制下能够在设备与主存储器之间直接传送数据,并且能够自动更新主存储器的缓冲区地址,自动为所传送的字符个数进行计数,直到一个数据块全部传送完成之后,才要求 CPU 进行服务。

4.1.3 基本输入输出方式

对于工作速度、工作方式和性质不同的外围设备,通常要采用不同的输入输出方式。目前常用的基本输入输出方式有如下三种。

4.1.3.1 程序控制输入输出方式

程序控制输入输出方式又称为状态驱动输入输出方式、应答输入输出方式、查询输入输出方式、条件驱动输入输出方式等。图 4.3 表示从键盘输入一个字符到处理机,再把这个字符输出到终端显示器上的工作过程。当 DONE 标志为“1”时,表示已经从键盘输入一个字符到设备缓冲寄存器,在这个字符被 CPU 取走后, DONE 标志复位。输出设备的标志 READY 的设置与输入设备正好相反。当 READY 标志为“1”时,表示设备缓冲寄存器

是空的,正准备接收从 CPU 送来的数据,在设备缓冲寄存器中已经有数据时,READY 标志复位,表示输出设备正在把缓冲寄存器中的数据输出到设备上。

程序控制输入输出方式有如下四个特点:

1. 何时对何设备进行输入或输出操作完全受 CPU 控制。
2. 外围设备与 CPU 处于异步工作关系。CPU 要通过指令对设备进行测试才能知道设备的工作状态,如工作已经结束(或称为设备空闲,已经准备就绪等),正在忙碌等。
3. 数据的输入和输出都要经过 CPU。外围设备每发送或接收一个数据都要由 CPU 执行相应的指令才能完成,如图 4.4 所示。
4. 用于连接低速外围设备,如终端、打印机等。

早期的计算机系统采用程序控制输入输出方式,一个处理机在一段时间内只能管理一台外围设备。例如,一台速度为 100MIPS 的计算机系统管理一台打印机。打印机的工作速度是每秒 10 个字符。处理机用一条指令就能向打印机传送 4 个字符。这样,处理机的利用率只有:

$$\frac{1}{100 \times 10^6 \times 4} = \frac{1}{4 \times 10^8}$$

即 4 千万分之一。



图 4.4 程序控制方式的数据传送过程

在现代计算机系统中,每台外围设备都设置有数据缓冲寄存器、状态寄存器(对输入设备)或控制寄存器(对输出设备)。这样,一个处理机就可以管理多台外围设备。如图 4.3 那样,一个处理机既可以管理键盘,又可以管理终端或打字机。

当一个处理机需要管理多台外围设备,而且这些外围设备又要并行工作时,处理机可以采用轮流循环测试方法,分时为各台外围设备服务。当然,程序流程不能像图 4.3 那样,要修改成如图 4.5 所示。当被测试的一台设备还在忙碌(或者还没有准备好)时,必须立即测试下一台设备。只要处理机对所有设备测试一个循环花费的时间小于最快设备的工作周期,那么,所有外围设备就都能够正确地并行工作。

采用程序控制输入输出方式的一个很明显的优点是灵活性很好。例如,可以很容易地改变各台外围设备的优先级。在图 4.5 中,程序员能够很方便地任意安排各台外围设备的测试顺序,而这个测试顺序实际上就是各台设备的优先级。

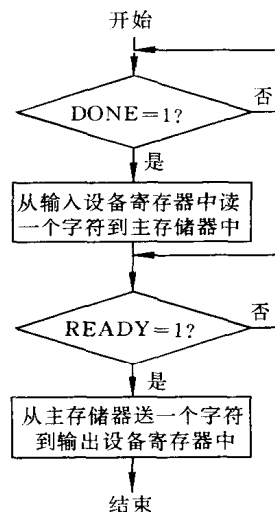


图 4.3 键盘输入和再显示的
程序框图

程序控制输入输出方式一个很大的缺点是：在一般情况下不能实现处理机与外围设备并行工作。

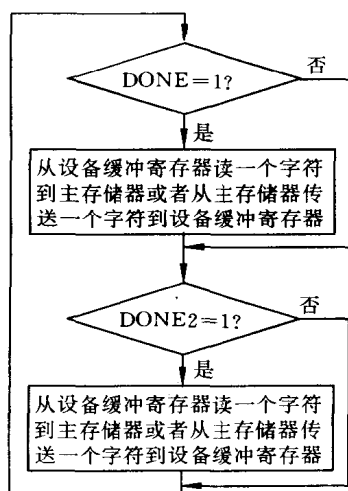


图 4.5 采用程序控制方式,一个处理机管理多台外围设备的程序框图

如果一定要实现处理机与外围设备并行工作的话,在图 4.3 中只能这样来管理:在把要输出的一个字符送到终端或打字机之后,处理机也可以先运行自己的一小段程序,然后再去测试键盘是否有输入字符。由于人敲击键盘的速度相对于处理机的运算速度来说是非常低的,因此处理机可以利用人两次敲击键盘的中间一段时间,运行自己的程序。这样,就可以实现处理机与两台外围设备并行工作。但是,这种管理方式实现起来很困难,而且也相当危险。主要原因是处理机运行程序这段时间的长短很难控制。如果这段时间太短,处理机的利用率很低;如果时间过长,超过了两次敲击键盘之间的这段时间,那么,从键盘上输入的第一个字符就将丢失。

4.1.3.2 中断输入输出方式

采用中断输入输出方式能够完全克服程序控制输入输出方式中处理机与外围设备之间不能并行工作的缺点。

为了实现中断输入输出方式,外围设备和 CPU 都必须增加相关的功能。在外围设备方面,要改变被动地等待 CPU 来为它服务的工作方式。当输入设备已经把数据准备就绪,或者输出设备已经空闲时,要主动向 CPU 发出服务的请求。在 CPU 方面,每当执行完成一条指令后都要测试有没有外围设备的中断服务请求。如果发现外围设备的中断服务请求,则要暂时停止当前正在执行的程序,先去为外围设备服务,等服务完成后再继续执行原来的程序。

中断输入输出方式的定义如下:

当出现来自系统外部,机器内部,甚至处理机本身的任何例外的,或者虽然是事先安排的,但出现在现行程序的什么地方是事先不知道的事件时,CPU 暂停执行现行程序,转去处理这些事件,等处理完成后返回来继续执行原先的程序。

中断输入输出方式的特点是:

1. CPU 与外围设备能够并行工作。
2. 能够处理例外事件,例如,电源掉电、非法指令、地址越界、数据溢出、数据校验错、页面失效等。

另外两个特点与程序控制输入输出方式相同。

3. 数据的输入和输出都要经过 CPU,要在程序的控制下完成从输入设备中读入数据到主存储器,或者把主存储器中的数据输出到输出设备中去。因此,中断输入输出方式与程序控制输入输出方式一样具有灵活性好的特点。

4. 一般用于连接低速外围设备。这是因为每输入或输出一个数据都必须执行一段程序才能完成。

在现代计算机系统中,中断输入输出方式的作用已经远远超出了为外围设备服务的范畴,成为现代计算机系统中非常重要的一个组成部分。如图 4.1 中,处理机与外部世界的联系大都是通过中断输入输出方式来实现的。在本章中有专门一节来介绍中断系统的工作原理和实现方法。

4.1.3.3 直接存储器访问(DMA)方式

直接存储器服务方式又称为 DMA(direct memory access)方式,这种输入输出方式主要用来连接高速外围设备,例如,磁盘存储器、磁带存储器等。

目前,一般高速活动头磁盘存储器的数据传输速度为每秒 30 兆字节以上,这个速度已经接近于采用动态随机存储器(DRAM)芯片实现的主存储器的工作速度。因此,对于这类高速外围设备,不能采用程序控制输入输出方式,也不能采用中断输入输出方式,必须在外围设备与主存储器之间建立直接数据通路。DMA 方式的数据传送过程如图 4.6 所示。因此,支持 DMA 方式的计算机系统必须采用以主存储器为中心的结构。

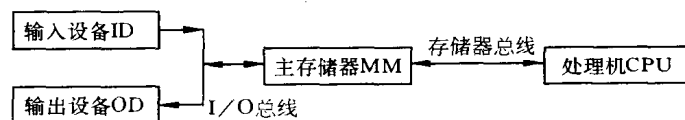


图 4.6 DMA 方式的数据传送过程

DMA 方式具有如下特点:

1. 主存储器既可以被 CPU 访问,也可以被外围设备访问。在主存储器中通常要有一个存储管理部件来为各种访问主存储器的申请排队,一般计算机系统把外围设备的访问申请安排在最高优先级。

2. 由于在外围设备与主存储器之间传送数据不需要执行程序,也不动用 CPU 中的数据寄存器和指令计数器等,因此,不需要做保存现场和恢复现场等工作,从而使 DMA 方式的工作速度大大加快。

3. 在 DMA 控制器中,除了需要设置数据缓冲寄存器、设备状态寄存器或控制寄存器之外,还要设置主存储器地址寄存器、设备地址寄存器和数据交换个数计数器。外围设备与主存储器之间的整个数据交换过程全部要在硬件控制下完成。另外,由于外围设备一般是以字节为单位传送的,而主存储器是以字为单位访问的,因此,在 DMA 控制器中还要有从字节装配成字和从字拆卸成字节的硬件。

4. 在 DMA 方式开始之前要对 DMA 控制器进行初始化,包括向 DMA 控制器传送主存缓冲区首地址、设备地址、交换的数据块的长度等,并启动设备开始工作。在 DMA 方式结束之后,要向 CPU 申请中断,在中断服务程序中对主存储器中数据缓冲区进行后处理。如果需要继续传送数据的话,要再次对 DMA 控制器进行初始化。

5. 在 DMA 方式中,CPU 不仅能够与外围设备并行工作,而且整个数据的传送过程

不需要 CPU 的干预。如果主存储器的频带宽度足够的话,外围设备的工作可以丝毫不影响 CPU 运行它自身的程序。

DMA 方式的工作流程如下。

对于输入设备:

从输入介质上读一个字节或字到 DMA 控制器中的数据缓冲寄存器 BD 中,如果输入设备是面向字符的,则要把读入的字符装配成字。

若一个字还没有装配满,则返回到上面;若校验出错,则发中断申请;若一个字已经装配满,则将 BD 中的数据送入主存数据寄存器。

把主存地址寄存器 BA(在 DMA 控制器中)中的地址送入主存地址寄存器,并且将 BA 中的地址增值至下一个字地址。

把 DMA 控制器内的数据交换个数计数器 BC 中的内容减“1”。

若 BC 中的内容为“0”,则整个 DMA 数据传送过程全部结束,否则返回到最上面继续进行。

对于输出设备:

把主存地址寄存器 BA(在 DMA 控制器中)中的地址送入主存地址寄存器,并启动主存储器,同时将 BA 中的地址增值至下一个字地址。

将主存储器数据寄存器中的数据送入 DMA 控制器的数据缓冲寄存器 BD 中。如果输出设备是面向字符的,则要把 BD 中的数据拆卸字符。

把 BD 中数据逐个字符(对于面向字符的设备)或整个字写到输出介质上。

把 DMA 控制器内的数据交换个数计数器 BC 中的内容减“1”。

若 BC 中的内容为“0”,则整个 DMA 数据传送过程全部结束,否则返回到最上面继续进行。

目前使用的 DMA 方式实际上有如下三种:

1. 周期窃取方式。在每一条指令执行结束时,CPU 测试有没有 DMA 服务申请,如果有,CPU 进入一个 DMA 周期。在 DMA 周期中借用 CPU 完成上面所列出的 DMA 工作流程,包括数据和主存地址的传送,交换个数计数器中的内容减“1”,主存地址的增值及一些测试判断等。

采用周期窃取方式时,主存储器可以不与外围设备直接连接,而只与 CPU 连接,即仍然可以采用如图 4.4 那样的连接方式,因为外围设备与主存储器的数据交换与程序控制输入输出方式和中断输入输出方式一样都是要经过 CPU 的。

周期窃取方式与程序控制输入输出方式和中断输入输出方式的不同处主要在:它不需要使用程序来完成数据的输入或输出,只是借用了 CPU 的周期来完成 DMA 流程。因此,其工作速度是很快的。

周期窃取方式的优点是硬件结构很简单,比较容易实现。缺点是在数据输入或输出过程中实际上占用了 CPU 的时间。

2. 直接存取方式,这是一种真正的 DMA 方式。DMA 控制器的数据传送申请不是发向 CPU,而是直接发往主存储器。在得到主存储器的响应之后,整个 DMA 工作流程全部在 DMA 控制器中用硬件完成。

直接存取方式的优点与缺点正好与周期窃取方式相反。

目前的多数计算机系统均采用直接存取方式工作。

3. 数据块传送方式。在设备控制器中设置一个比较大的数据缓冲存储器,一般要能够存放下一个数据块,如在软磁盘存储器中通常设置一个 512 个字节的数据缓冲存储器。与设备介质之间的数据交换在数据缓冲存储器中进行。设备控制器与主存储器之间的数据交换以数据块为单位,并采用程序中断方式进行。

数据块传送方式实际上并不是 DMA 方式,只是它在每次中断输入输出过程中是以数据块为单位获得或发送数据的,这一点与上面两种 DMA 方式相同,因此,通常也把这种输入输出方式归入 DMA 方式。

采用数据块传送方式的外围设备还有行式打印机、激光打印机、卡片阅读机、部分绘图仪等。

4.2 中断系统

在现代计算机系统中,中断系统已经不仅仅属于输入输出系统。它除了被用来管理各种各样的外围设备之外,在整个计算机系统起着重要的作用。例如,人机联系、故障处理、实时处理、多任务操作系统、分时操作系统、程序的跟踪调试、程序的监测、用户程序与操作系统的联系、多处理机系统中各个处理机之间的相互联系及任务分配等。本节并不介绍中断系统的工作原理,而主要介绍在中断系统设计过程中软件与硬件的功能分配及与其相关的一些问题。

4.2.1 中断源的组织

引起中断的各种事件称为中断源。中断系统的复杂性实际上是由中断源的多样性引起的。中断源可以来自系统外部,也可以来自机器内部,甚至处理机本身。中断可以是硬件引起的,也可以是软件引起的。把各种各样的中断源分类、分级组织好,是设计中断系统时首先要做好的一件事情。

4.2.1.1 中断源的种类

常见的中断源有如下几种类型:

1. 由外围设备引起的中断。通常包括:低速外围设备的数据缓冲寄存器已经准备好接收或发送数据,高速外围设备采用 DMA 方式完成一个数据块传送之后的处理工作,外围设备的启动和停止,完成对外围设备的控制(如磁盘和磁带存储器的定位等),输入输出过程中任意一个环节出现错误等。

2. 由处理机产生的中断。例如,算术运算操作溢出,除数为零,数据校验错,非法数据格式等。响应中断后进入监控程序,处理机处于动态停机状态,等待操作员给予处理。或者返回到用户程序,等待用户修改程序或数据。

3. 由存储器产生的中断。例如,非法地址(包括地址越界、地址不存在、写 ROM 地址),动态随机存储器(DRAM)刷新,主存储器页面失效,数据或地址校验错,访问主存储

器超时错等。

4. 由控制器产生的中断。例如,非法指令,未定义的操作码,用户程序执行了特权指令,堆栈溢出,分时系统中时间片到,操作系统用户态与特权态的切换等。

5. 由总线产生的中断。包括输入输出总线出错,存储器总线出错等。

6. 实时过程控制产生的中断。例如,实时检测设备的采样中断,为某些实时控制设备发送控制信号等。

7. 实时钟的定时中断。

8. 多处理机系统中,从其他处理机发送来的中断,控制台开关中断等。

9. 程序调试过程中,执行完一条指令或程序运行到一个事先设置的断点时,通过中断进入监控程序,以便对被调试程序进行跟踪或监测。

10. 硬件故障中断。通过监控程序调用诊断程序对机器各个部分进行诊断,如果诊断没有错误则重新引导机器,否则停机。

11. 电源故障中断。这时必须停止其他一切工作,保存处理机全部状态信息和挥发性存储器中的内容。

4.2.1.2 中断源的分类组织

在现代计算机系统中,中断源的数目很多,一般有几个至几百个。为了在响应中断后处理机能够尽快找到中断入口,以便为中断源提供服务,必须对这些中断源进行分类。通常根据中断事件的紧迫程度、中断源工作速度的高低、中断源的性质等进行分类。对每一类中断源分配一个硬件的中断入口,在进入这个入口之后,再通过软件找到具体的中断源。

IBM 公司的机器通常把中断源分为 6 类:

1. 重新启动中断。这是为操作人员重新启动一个程序用的,在一般情况下,处理机不能禁止这类中断。

2. 机器检验出错中断。当发生硬件或软件故障时产生。用一个 64 位的机器检验中断码保存中断的原因及其严重程度。在机器的检验保存区中还存储有更加详细的中断原因和故障位置的说明。机器检验出错中断主要包括有电源故障、运算器误动作、主存储器校验错、输入输出通道硬件故障及处理机的其他各种故障等。

3. 程序性错误引起的中断。包括指令或数据格式错误、在程序执行过程中出现非法操作码、主存保护错误、地址越界错误、各种运算溢出错误、除数为零错误、有效位为零错误、用户态下使用管态指令错误等。另外,还有程序的事件记录、监督程序对事件的检测引起的中断等。

4. 访问管理程序中断。当用户程序要调用管理程序时,执行访管指令引起的中断。处理机一般不能禁止这类中断。

5. 外部事件中断。事件可以来自机器外部,也可能来自机器内部。包括各种用于计时、计费、控制的定时器中断,各种用于与其他机器和系统联系的外来信号中断,用于操作员对机器进行干预的中断键的中断。

外部事件中断又分为两类,一类中断在没有得到处理机响应时能继续保留,而另一类

中断如果处理机不响应则不再保留。

6. 输入输出中断。用于处理机管理各种外围设备,管理通道处理机等。

对于后四类中断源各有一个 16 位的中断码,这个中断码用来区分各个具体的中断源。当处理机响应中断,从硬件入口进入各类中断源之后,可以通过这个中断码来找到是哪个中断源发出的中断请求。

有许多机器把中断源分为可屏蔽中断和不可屏蔽中断两大类,或称为一般中断和异常中断(exception interrupt)。对于不可屏蔽中断,不能通过软件屏蔽它,它一旦申请中断服务,处理机必定会响应。对于可屏蔽中断,可以通过软件把它屏蔽掉。例如,如果不希望某一台打印机工作,可以通过一条指令向这台打印机发一个中断屏蔽码。在这以后,即使这台打印机申请中断服务,这个中断申请信号也不能送往处理机。

有的机器按照中断事件的紧迫程度来划分可屏蔽中断和不可屏蔽中断,例如,把电源掉电、机器硬件故障等划分为不可屏蔽中断。

在 IBM 370 系列机中,把执行现行指令引起的中断划分为不可屏蔽中断。例如,运算结果溢出、主存页面失效等。这类中断一般不能被屏蔽,一旦出现,处理机必须响应并给予处理,否则这些异常的中断请求将会因被屏蔽而被丢失,造成程序无法继续运行下去或者程序运行错误。可屏蔽中断是指那些与当前进程无关的中断事件,如机器硬件故障引起的中断请求、外围设备的中断请求、定时器的中断请求等。这些中断可以被屏蔽,没有得到处理机响应的中断请求被保存在中断寄存器中不会被丢失,当屏蔽被解除之后,仍然能够继续得到响应和处理。

在异常中断中,有一类称为自陷(trap)中断,它的中断请求发生在一些特殊指令的末尾,经中断服务程序处理后返回到正常执行程序的下一条指令继续执行。另一类称为故障(fault)中断,它的中断请求可能发生在任何一条指令的执行过程中,经中断服务程序处理后,要返回到原先发生故障的那条指令处重新执行引起故障的那条指令。还有一类称为失效(abort)中断,它的中断请求也可能发生在一条指令的执行过程中,但是,除非强制干预或系统重新复位,否则机器无法继续正常工作下去。

4.2.1.3 中断优先级

中断源的中断请求一般是随机的,在中断源比较多的情况下,很可能同时发生多个中断请求。CPU 必须安排一个响应和处理中断的优先顺序。中断优先级的确定是一个涉及计算机系统全局的问题,主要由下列因素来决定:

1. 中断源的紧迫性。如电源故障、总线错、CPU 的地址错、数据错等,这些机器检验性错误引起的中断一般要安排在最高优先级。它们一旦出现,必须及时处理,否则整个系统都将无法正常运行。而那些仅影响局部的故障,其优先级可以安排在低一些的级别,如程序性错误引起的中断请求,外围设备的输入输出中断请求等。

2. 设备的工作速度。快速设备由于数据存在的时间短,必须及时响应以避免数据丢失,其优先级应安排得高一些。例如,在一些常用的外围设备中,优先级从高到低的次序一般如下:

实时钟

磁盘存储器,包括软磁盘

行式打印机

控制台终端输出

控制台键盘输入

3. 数据恢复的难易程度。数据丢失后无法恢复的设备,其优先级应高于能自动或手动恢复数据的设备。

4. 要求处理机提供的服务量。能够大部分时间独立工作而较少要求处理机干预的事件,其优先级应高于需要处理机连续为它服务的事件。两个极端的例子是:DMA 请求和执行主程序。

在 IBM 370 系列机中,把 7 类中断分为 5 级,从高到低分别是:

重新启动引起的中断;紧急的机器检验错误引起的中断;程序性错误、调用管理程序、可以抑制的机器检验错误引起的中断,这三类中断是互斥的,不可能同时发生;外部事件引起的中断;外围设备的中断。

在 DEC 公司的机器中,中断优先级从高到低分别是:

总线错误引起的中断;主存刷新中断;指令错误引起的中断;程序跟踪中断;电源掉电中断;在线停机中断;在线事件中断,如实时钟等;外围设备的中断;用户程序中断。

有的计算机系统中还设置有 0 级中断。当机器故障重叠发生或无法排除,系统完全不能正常工作时,由中断系统的硬设备发出机器报警信号。对于单处理机,这种机器故障一般只有经操作人员的干预才能排除。在多处处理机中,可以进行机器间的任务切换。这种告急状态虽然也使机器所执行的程序被中断,但它并不是真正的中断级,它并不参与中断优先级的排队,中断发生后也无法自行恢复。

在现代计算机系统中,中断优先级一般是由硬件的排队器实现的,因此,当有多个中断源同时请求中断服务时,中断响应次序的高低是固定死的。优先级高的中断源先被处理机响应。处理机在执行某一个级别的中断源的中断服务程序时,与它同级的或比它低级的中断源的中断请求不能中断它的服务程序,只有比它高级的中断源的中断请求才能中断其服务程序。这时,处理机响应这个高级的中断源的中断请求,转去为它服务,待服务完成后,再返回来继续执行原先的那个中断服务程序。

下面举一个例子来说明不同优先级的中断源随机请求中断服务时,它们的中断响应和中断服务的过程。

共有 4 个中断源,中断优先级从高到低分别是:1 级、2 级、3 级和 4 级。当处理机执行主程序时,同时有 3 级中断源和 2 级中断源向处理机发出中断服务的请求。因为 2 级中断源的中断优先级高于 3 级中断源,因此它首先得到处理机的响应,并进入其中断服务程序。这时,虽然又有 4 级中断源发出中断服务请求,但是,由于在等待中断服务的 3 级中断源和 4 级中断源的中断优先级都低于处理机正在为它服务的 2 级中断源,因此,它们都不能中断 2 级中断源的中断服务程序。只有当 2 级中断源的中断服务过程全部执行完成,处理机返回来执行主程序时,再次在等待中断服务的 3 级和 4 级两个中断源中选择一个中断优先级高的 3 级中断源。处理机响应 3 级中断源的中断请求,并为它服务。待 3 级中断

源的中断服务完成,处理机又返回执行主程序时,才能响应 4 级中断源的中断请求,并为其服务。处理机在执行 4 级中断源的中断服务程序的过程中,1 级中断源发出中断请求。由于 1 级中断源的优先级高于 4 级中断源,因此,它可以中断 4 级中断源的中断服务程序。处理机直接从 4 级中断源的服务程序中转移去执行 1 级中断源的中断服务程序,待 1 级中断源的中断服务程序全部执行完成后再返回到 4 级中断源的中断服务程序中继续执行。直到 4 级中断源的中断服务程序全部完成并返回到主程序,整个中断响应和中断服务过程才全部结束。

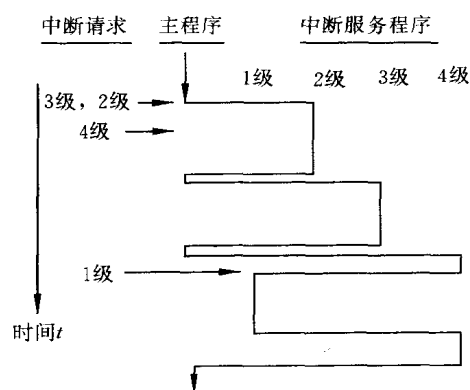


图 4.7 按照中断优先级响应中断请求的例子

4.2.2 中断系统的软硬件功能分配

从中断源发出中断服务请求,到这个中断服务请求被处理机响应并全部处理完成,其过程是相当复杂的。其中,有些功能必须用硬件来实现,有的功能必须用软件来实现,而中间的大部分功能既可以用硬件实现,也可以用软件实现。因此,在设计一台计算机的中断系统时,如何恰当地分配中断系统的软件与硬件功能,是设计好中断系统最关键的一个问题。

中断系统中软件与硬件的功能分配主要考虑如下两个因素:

1. 中断响应时间

从某一个中断源发出中断服务请求到处理机响应这个中断源的中断服务请求,并开始执行这个中断源的中断服务程序所用的这一段时间称为中断响应时间。

在中断系统中,中断响应时间是一个非常重要的指标。特别是在实时计算机系统中,中断响应时间是整个计算机系统的一个关键性指标。

2. 灵活性

一般情况下,用硬件实现速度快,但灵活性差。用软件实现正好相反,灵活性好,但速度低。

上述这两个要求实际上是互相矛盾的。如果要减小中断响应时间,那么,中断处理过程中那些既能用硬件实现,也能用软件实现的功能,要尽量用硬件来实现,但是这样做就必然失去了灵活性。相反,如果用软件实现的功能多了,灵活性虽然好了,但中断响应时间

就必然要增加。

4.2.2.1 中断处理过程

从某一个中断源发出中断服务请求,到这个中断服务请求全部处理完成所经过的主要过程如图 4.8 所示。

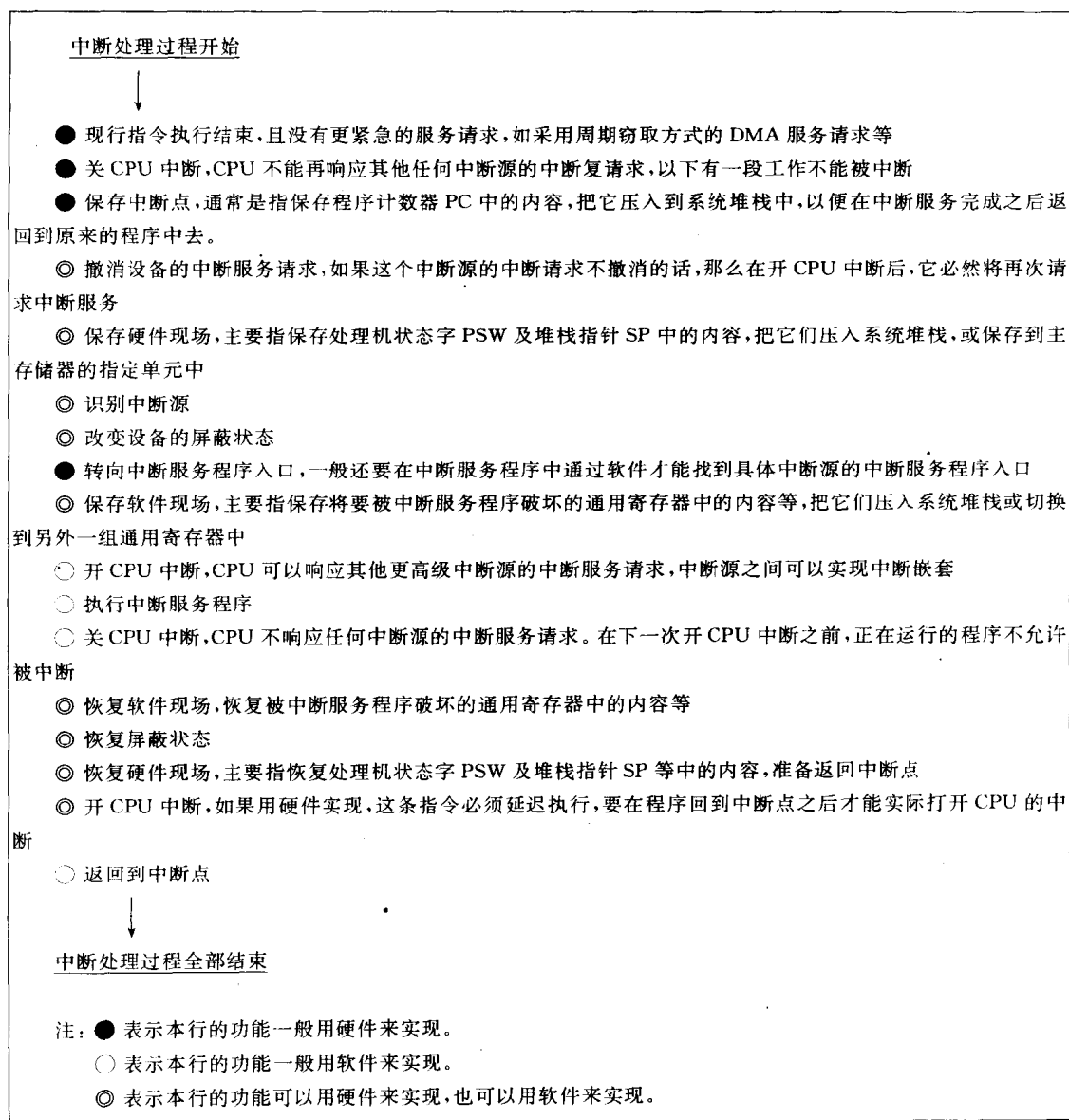


图 4.8 中断处理过程

图 4.8 中给出的中断处理顺序在不同的计算机系统中可能有所不同。例如,“转向中断服务程序入口”这一功能可以插在从“保存中断点”之后,到第一次“开 CPU 中断”之间的任何一个地方。具体插在什么地方要看中间的一些功能,如“撤消设备的中断服务请

求”、“保存硬件现场”、“识别中断源”、“改变设备的屏蔽状态”和“保存软件现场”等是用硬件来实现,还是用软件来实现。一般要在采用硬件实现的功能全部结束之后,才能执行“转向中断服务程序入口”这一功能。用软件实现的所有功能都在中断服务程序中完成。

实际上,在图 4.8 所列的全部功能中,除了“现行指令执行结束”之外,只有“保存中断点”和“转向中断服务程序入口”这两个功能是必须用硬件来实现的。这是因为中断响应发生在现行程序的什么地方是不确定,一般不能由程序员来安排;因此,这两个功能必须由硬件来做。另外,第一次“关 CPU 中断”一般也用硬件来实现,它只要用很简单的组合逻辑就能实现,几乎不需要花费时间。而“保存中断点”和“转向中断服务程序入口”这两个功能,实际上可以隐含地执行一条子程序调用指令来实现。

同样,也只有“执行中断服务程序”和“返回到中断点”这两个功能是必须用软件来实现的。其中,“返回到中断点”可以通过执行一条中断返回指令来实现,就像从子程序中返回到主程序那样。至于“执行中断服务程序”当然是要用软件来实现的,否则也就不能称为“中断”了。

4.2.2.2 中断响应时间

从一个中断源向处理机发出中断服务请求开始,到处理机实际开始执行这个中断源的中断服务程序时为止,这一段时间称为中断响应时间。

如果中断响应时间过长,在实时控制系统中,很可能失去控制的时机或丢失控制信号。在数据采集或数据传输系统中,有可能丢失数据。

影响中断响应时间的因素主要有四个:

1. 最长指令执行时间。在一条指令执行期间,一般不允许被中断。在有些计算机系统,特别是 CISC(复杂指令系统计算机)中,有些指令的执行时间很长,甚至无法预测。例如,主存储器中的数据块“搬家”指令,数据匹配指令,向量运算指令等。对于这些执行时间特别长的指令,要在指令中间设置允许响应中断的时刻。为了实现这一点,在处理机内也必须采取相应的措施,例如,要增加一个指令寄存器,用来保存还没有执行完的指令内容等。

由于一个中断源的中断请求是随机发出的,可能发生在一条指令执行过程中的任何时刻。因此,考虑最坏情况,是最长指令的执行时间。

2. 在一条指令执行完成后,处理其他更紧急的任务所用时间。例如,采用周期窃取方式工作的 DMA 服务请求等。如果 DMA 采用直接存储器存取方式,则在一般计算机系统中不需要这一段时间。

3. 从第一次“关 CPU 中断”到第一次“开 CPU 中断”所经历的时间。这一段时间内要做的事情比较多。从图 4.8 中看,共 7 件事情要做,再加上开和关 CPU 中断本身,则共要做 9 件事情。

在整个中断响应时间中,这一段时间往往是最主要的。如果这些要做的事情都用硬件来完成,中断响应时间就可以缩短很多。相反,如果其中的大部分功能都用软件来实现,则中断响应时间就会很长。

中断系统的软件与硬件功能分配,主要就是要考虑这一段时间内所要做的事情用软

件来实现,还是用硬件来实现。

4. 多个中断源同时请求中断服务时,通过软件找到相关中断源的中断服务程序入口所经历的时间。

上述四部分时间中,最主要的是第1和第3两部分。其中,第1部分时间是指令系统设计时考虑的问题,在中断系统的设计中,主要考虑第3部分的时间。

分析从第一次“关CPU中断”到第一次“开CPU中断”之间所要做的7件事情,它们中有5件是既可以用软件来实现,也可以用硬件来实现的。其中,“撤消设备的中断服务请求”只需要用很简单的组合逻辑就能实现,因此,在一般计算机系统中都用硬件直接实现。其他的4件事情将分别在下面的各节中讨论。其中,把“保存硬件现场”和“保存软件现场”两件事情放在一起介绍。

4.2.2.3 识别中断源的查询法

识别中断源最简单的一种方法是查询法(poling method)。这种方法所需要的硬件非常简单,几乎全部用软件来实现。

如图4.9所示,采用查询法来识别中断源的过程分如下三步:

第一步:所有中断源公用一条中断请求线,一般可直接采用“线或”的办法来实现。无论是哪一个,也不管有多少个中断源发出中断服务请求,得到处理机响应后都进入同一个中断服务程序入口。

第二步:转入公共的中断服务程序入口。方法很简单,在得到处理机的中断响应之后,隐含执行一条无条件间址指令:

JMP @INTR

在主存储器的INTR单元中存放中断服务程序的入口地址。

第三步:用软件逐个测试中断源的状态。凡是发出中断服务请求的中断源,它的完成标志位(DONE)必然被置位,而忙标志(BUSY)一般被清除。测试的顺序实际上就是中断源的中断优先级。因此,查询法又被称为跳步链程序法(skip-chain program method)或依次测试法。

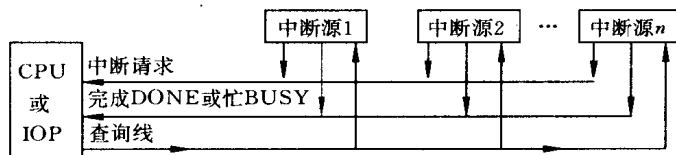


图 4.9 识别中断源的询问法

例如,有打印机、键盘输入和终端显示三个中断源,在进入公共的中断服务程序之后,可以通过执行如下的一小段中断测试程序,很容易地找到这三个中断源的中断服务程序入口:

```
SKIP DZ, PRN      ; 测试打印机 PRN 的完成标志 DONE 是否为“0”,
                  如果 DONE=0,跳过下一条指令,否则继续执行下一条指令。
```

```

JMP PRINT      ;打印机的 DONE=1,转打印机的中断服务程序入口 PRINT
SKIP DZ, KEY   ;测试键盘输入 KEY 的完成标志 DONE=0?
JMP KEYBORD    ;键盘输入的 DONE=1,转它的中断服务程序入口 KEYBORD
SKIP DN, MNT   ;测试终端显示 MNT 的完成标志 DONE=1?
JMP ERROR      ;所有中断源均没有请求中断服务,转错误处理程序入口 ERROR
JMP MONITOR    ;终端显示 MNT 的 DONE=1,转它的中断服务程序入口 MONITOR

```

查询法的主要优点是灵活性好。因为用程序对设备进行测试的顺序实际上就是中断源的中断优先级,而测试顺序是程序员可以通过软件任意改变的。主要的缺点是速度太慢,特别是在中断源比较多的时候。

4.2.2.4 识别中断源的串行排队链法和中断向量法

为了提高识别中断源的速度,必须用硬件来完成中断源的排队,这种方法称为串行排队链法(daisy-chain method)。具体实现方法有如下三种:

1. 设置一个中断请求寄存器。每个中断源在中断请求寄存器中占据相应的一位,并且按照中断的优先级从高到低的顺序排列。当一个中断源请求中断服务时,中断请求寄存器中的相应位置“1”,否则相应位为“0”。

所有中断源使用一条公共的中断请求线,找到中断源服务程序入口的过程,第一步和第二步与上面的查询法相同。在处理机响应中断源的中断请求,并转入公共的中断服务程序之后,用一条专门的指令把中断请求寄存器中的内容读到通用寄存器中,然后用一条按位扫描指令找到第一位为“1”的位号,这个位号实际上就是所有请求中断服务的中断源中,具有最高优先级的中断源的相对编号。然后通过执行一条变址转移指令就能直接转移到这个中断源的中断服务程序入口。一个简单的程序例子如下:

```

INTA R1      ;把中断请求寄存器中的内容读入到通用寄存器 R1 中
SBT R1 R2    ;按位扫描 R1 寄存器,得到请求中断服务的具有最高优先级
              的中断源的相对编号,结果放在 R2 寄存器中
JMP @VTAB(R2) ;转向中断源的中断服务程序入口(VTAB+(R2))
VTAB: DEV1   ;第一个中断源(最高优先级)的中断服务程序入口
      DEV2   ;第二个中断源的中断服务程序入口
      ...
      DENn   ;第 n 个中断源的中断服务程序入口

```

这种方法只要执行三条指令就能够找到所需要的中断源的中断服务程序入口。如果同时有多个中断源都请求中断服务,则找到的是所有请求中断服务的中断源中具有最高优先级的中断源。如果主存储器是按字节编址的,而主存储器的字长是 32 位的话,则 R2 中的中断源相对中断优先级还要乘以 4,即左移两位。这种方法完全避免了跳步链程序法需要逐个中断源进行测试的缺点。

当中断源很多时,还可以设置两级或多级中断请求寄存器。这时,要把中断源逐级分组。只有最低一级中断请求寄存器中的每一位对应一个中断源的中断服务请求。再高一级的中断请求寄存器,每一位对应一组中断源的中断服务请求。当然,这种方法扫描中断

源寄存器的过程要复杂些。

2. 上面的方法是用软件与硬件相结合的办法,最终通过一条按位扫描指令找到所有请求中断服务的中断源中,具有最高优先级的中断源的相对编号的。为了进一步加快识别中断源的速度,可以用硬件的串行排队器和编码器来完成这一工作。中断源与处理机的连接关系如图 4.10 所示。

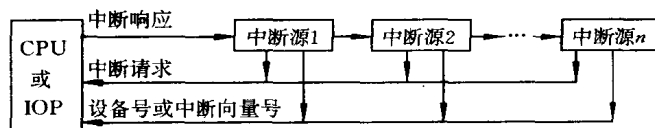


图 4.10 识别中断源的串行排队链法

硬件的排队器分布在各个中断源中,编码器可以集中放在处理机中。一个有四个中断源的排队器和编码器如图 4.11 所示。

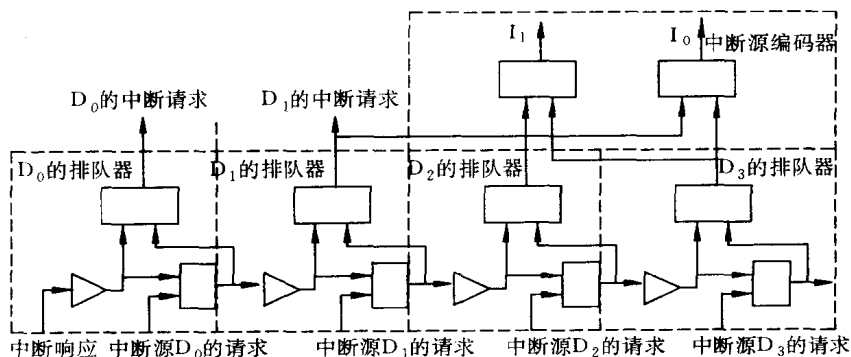


图 4.11 四个中断源的中断排队器和编码器

图 4.11 中,长方形为与非门,三角形为反向器。处理机发出的中断响应信号和各个中断源发出的中断请求信号都以低电平有效。中断源编码器的输出信号 I_1I_0 就是所有请求中断服务的中断源中,优先级最高的中断源的编号。例如,只要有 D_0 中断源请求中断服务,则无论其他中断源是否发出中断服务请求, I_1I_0 都输出 00。同样,当 D_0 中断源不请求中断服务, D_1 中断源请求中断服务时,则无论 D_2 和 D_3 中断源是否发出中断服务请求, I_1I_0 都输出 01。只有当 D_0 、 D_1 和 D_2 中断源都不请求中断服务, D_3 中断源请求中断服务时, I_1I_0 才输出 11。

识别中断源的过程,第一步和第二步与上面的方法相同。在处理机响应中断源的中断服务请求,并转入公共的中断服务程序之后,用一条专门的指令就能直接读到所有请求中断服务的中断源中,具有最高优先级的中断源编号。然后同样通过执行一条变址转移指令转移到中断源的中断服务程序入口:

INTA R1 ;得到所有请求中断服务的中断源中,具有最高优先级的
的中断源编号,结果放在 R1 通用寄存器中


```

        JMP @VTAB(R1) ;转向中断源的中断服务程序入口(VTAB+(R1))
VTAB: DEV1             ;D0 中断源(最高优先级)的中断服务程序入口
      DEV2             ;D1 中断源的中断服务程序入口
      ...
      DENn            ;Dn 中断源的中断服务程序入口

```

这种方法由于使用了硬件排队器和编码器,因此识别中断源的速度更快。在进入公共的中断服务程序入口之后,只需要执行两条指令就能转入到请求中断服务的具有最高优先级的中断源的中断服务程序入口中。

3. 中断向量法(vectored interrupt method)。这是一种识别中断源速度更快,使用也更为广泛的方法。

中断向量法把上面第二种方法中的全部工作都用硬件来实现,包括用来识别中断源的两条指令。它的主存储器的固定区域中开辟出一个专用的中断向量区,同样用硬件排队器和编码器在所有请求中断服务的中断源中,产生具有最高优先级的中断源编号,然后隐含执行上面方法中的两条识别中断源的指令,直接通过硬件转向这个中断源的中断服务程序入口。

中断向量法不需要进入公共的中断服务程序,从而能够实现向中断服务程序入口地址的最快转移。

上面三种识别中断源的方法,都是通过硬件固定了各个中断源的中断优先级,并采用串行排队链来识别中断源的;因此,它们都属于串行排队链法。这类方法与查询法相比,节省了用程序来逐个测试中断源的时间,因此,中断源的识别速度更快。特别是中断向量法,其识别中断源的速度非常快。另外,由于串行排队器是分布在各个中断源的接口部件中的,各个中断源与处理机的连线很少,实现比较简单。但是,串行排队链法有两个明显的缺点。第一、各个中断源的中断优先级是由硬件固定死的,不能由程序员通过软件来改变,因此,它的灵活性比较差。第二、由于排队链是串行,而且分布在各个中断源中,只要其中任何一个中断源出现故障,整个串行排队链就都不能正常工作,因此,它的可靠性比较差。

4.2.2.5 识别中断源的独立请求法

为了克服串行排队链法可靠性差的缺点,提出了独立请求法(independent request method),如图 4.12 所示。

独立请求法的基本原理是:各个中断源使用自己独立的中断请求线 INIR,每一根中断请求线在处理机中都有固定的或可编程的中断优先级。如果同时有多个中断源请求中断服务,处理机可以通过它的仲裁线路立即选择其中具有最高优先级的中断源,向它发出中断响应信号 INIT,中断源用 INIT 信号清除它的中断请求信号,处理机就可以立即转入这个中断源的中断服务程序。这样,不需要用软件或硬件对中断源进行扫描,也不需要中断源回送中断源编号或中断向量等。

与串行排队链法相比,独立请求法实际上是把分布在各个中断源内的串行排队器都集中到处理机中,从而克服了串行排队链法可靠性差的缺点,但灵活性差的缺点仍然存

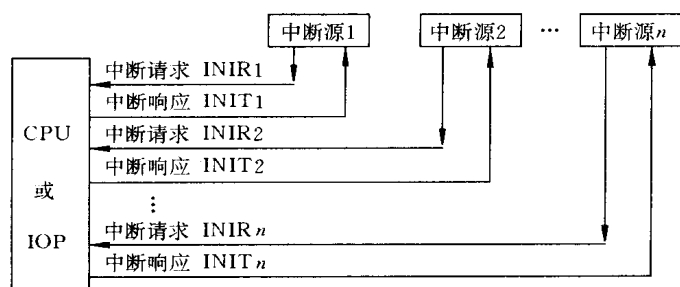


图 4.12 识别中断源的独立请求法

在。另外,中断源与处理机的连线比较多。因此,在中断源很多时,要把独立请求法与串行排队链法结合起来。如图 4.13 所示,把中断源分组,组内采用串行排队链法,组间采用独立请求法。这种方法称为分组独立请求法。

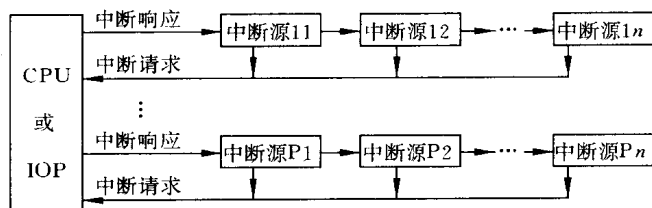


图 4.13 识别中断源的分组独立请求法

在 IBM 370 系列机中,就是采用分组独立请求法来识别中断源的。它把所有中断源分成六类,每一类都在主存储器的固定地址单元中分别存放有新的和旧的处理机状态字 PSW,如表 4.1 所示。

表 4.1 IBM 370 系列机中断源分类情况

中断源种类	旧 PSW 地址	新 PSW 地址	屏蔽位	中断码
机器检验错	48	112	13	与机器有关
调用管理程序	32	96	不可屏蔽	24~31 位
内中断	40	104	只能对以下四种屏蔽	24~31 位
定点溢出			36	
十进制溢出			37	
阶码溢出			38	
有效位为零			39	
外部事件中断	24	88	7	16~31 位
输入输出中断	56	120		
通道 0~5			0~5	16~23 位通道地址
通道 6 以上			6	24~31 位设备地址

处理机响应中断后,按照中断源的种类把处理机当前的状态字 PSW 存入到主存储

器这类中断源的旧 PSW 地址单元中,同时从相应的新 PSW 地址单元中取出新的处理机状态字。新 PSW 中的指令地址字段(第 40~63 位)就是这类中断源的中断服务程序的入口地址。退出中断服务程序时,通过再次交换处理机状态字 PSW,就能返回到中断点,处理机可以继续执行原来的程序。

4.2.2.6 中断现场的保存和恢复

中断现场的保存和恢复分别是中断处理机过程开始和结束时必须执行的步骤。现场信息可分为三类:

第一类,主要指程序计数器 PC 中的内容,它必须由硬件来完成保存,可以保存到主存储器的固定单元中,也可以压入系统堆栈。在中断服务程序结束前,通过执行一条中断返回指令来恢复。如果采用保存到主存储器固定单元中的办法,则在发生中断嵌套时,要通过程序来进行处理。

第二类,是指记录当前程序状态的有关信息,包括处理机状态字、堆栈指针、基址寄存器、中断屏蔽码等。这些信息是与当前正在运行的程序相关的,在转入中断服务程序后,这些信息必然要被破坏,因此必须加以保存。在图 4.8 中把这些信息称为硬件现场。为了缩短中断响应时间,许多机器采用硬件来保存这些信息。具体方法有如下三种:

1. 保存到主存储器的固定区域中。这种方法的缺点是:在发生中断嵌套时,要通过中断服务程序来进行处理,通常是搬到主存储器的另一个区域中,或者暂时存放在通用寄存器中。如果这个中断服务程序是为最高优先级的中断源服务的,则可以不处理这些信息的保存工作。

2. 压入系统堆栈。有堆栈的计算机系统,大多采用这种方法。

3. 交换处理机状态字。如 IBM 370 系列机,把与当前运行的程序有关的全部信息都存放在一个 64 位的处理机状态字中。在主存储器的固定区域中为每一类中断源开辟出两个 64 位的存储区域,其中一个用来存放将要执行的中断服务程序的状态字,另外一个则存放当前程序的处理机状态字。通过交换处理机状态字来实现现场的保存和恢复,其中还包括程序计数器 PC。

当然,如果为了节省硬件,这些与当前运行程序有关的硬件现场也可以采用软件在中断服务程序中加以保存和恢复。

第三类,是指在中断服务程序中将要被破坏的通用寄存器中内容。在图 4.8 中,把它称为软件现场。对于软件现场,大多数机器都采用软件来保存和恢复,即在中断服务程序的开始保存在中断服务程序中将要用到的那些通用寄存器中的内容,在中断服务程序结束前恢复这些通用寄存器中的内容。

为了加快保存和恢复软件现场的过程,在一般机器中都设置有成组存和成组取的指令,即可以用一条指令就能够完成保存或恢复所有通用寄存器中内容的工作。在第二章中曾经介绍过这类指令。

也有些机器采用硬件来保存软件现场。例如,SUN 公司的 Sparc 处理机,内部有 8 套通用寄存器,每一级中断源可以分别使用其中的一套通用寄存器。

4.2.3 中断屏蔽

上一节中介绍过,为了缩短中断响应时间,目前大多数机器都采用硬件来识别中断源,如串行排队链法、中断向量法、独立请求法和分组独立请求法等。这些方法虽然识别中断源的速度很快,但是,由于中断源的中断优先级是由硬件固定死的,不能由程序员通过软件来改变,因此灵活性比较差。采用中断屏蔽方法可以部分地解决快速识别中断源和灵活性之间的矛盾。

一般的方法是为每一个中断源设置一个中断屏蔽位。这些中断屏蔽位可以分布存放在各个中断源中,也可以集中存放在处理机内。另外,处理机要设置专门的指令来管理这些中断屏蔽位。当中断屏蔽位为“1”时,表示对应的中断源不能请求中断服务,为“0”时,对应的中断源可以请求中断服务。

设置中断屏蔽有如下三个用处:

1. 在中断优先级已经由硬件确定了的情况下,改变中断源的中断服务顺序。实际上,当有多中断源同时请求中断服务时,处理机响应中断请求的顺序是:在没有被屏蔽的中断源中,找出优先级最高的一个中断源先响应。因此,可以通过在一段时间内屏蔽掉较高级中断源的中断请求,让处理机先为较低级的中断源服务,等较低级的中断源的中断服务程序执行完成后,再解除较高级中断源的屏蔽,使处理机为这个较高级中断源服务,从而达到在有多中断源同时请求中断服务时,处理机先为较低级中断源服务,然后再为较高级中断源服务的目的。

2. 决定设备是否采用中断方式工作。在前面介绍过,外围设备有三种基本的输入输出方式。通过中断屏蔽,可以让某些外围设备不采用中断方式工作,而采用程序控制方式或DMA方式工作。例如,在向量计算机中,当一个很长的向量正在运算时,一般不希望被外围设备频繁地中断。

3. 在多处理机系统中,可以通过中断屏蔽,把对外围设备的输入输出服务工作分配到各个处理机中。

中断屏蔽的实现方法主要有两种:

第一种、每个或每级中断源设置一个中断屏蔽位的方法

中断屏蔽位可以分布在各个中断源中,也可以集中在处理机中,例如放在处理机的状态字中。这是一种用得比较普遍的方法。从表4.1中可以看到,IBM 370系列机就采用了这种方法。

下面,举一个具体的例子。

有四个中断源 D_1 、 D_2 、 D_3 和 D_4 ,它们的中断优先级从高到低分别是1级、2级、3级和4级。这些中断源的正常中断屏蔽码和改变后的中断屏蔽码见表4.2,每个中断源一位,共4位屏蔽码。

如果在4个中断源的中断服务程序中都使用正常的中断屏蔽码,处理机的中断服务顺序将严格按照中断源的中断优先级进行。图4.7就是这样的例子。当具有某一个中断优先级的中断源占有处理机,正在执行它的中断服务程序时,只有更高一级中断源的中断服务请求才能中断这个中断服务程序,同一级的和比它低级的所有中断源的中断服

务请求都不能中断当前的中断服务程序。

表 4.2 四个中断源的中断优先级和屏蔽码

中断源名称	中断优先级	正常中断屏蔽码				改变后的中断屏蔽码			
		D ₁	D ₂	D ₃	D ₄	D ₁	D ₂	D ₃	D ₄
D1	1	1	1	1	1	1	0	0	0
D2	2	0	1	1	1	1	1	0	0
D3	3	0	0	1	1	1	1	1	0
D4	4	0	0	0	1	1	1	1	1

如果采用改变后的中断屏蔽码,当 D₁、D₂、D₃ 和 D₄ 这 4 个中断源同时请求中断服务时,处理机实际为各个中断源服务的先后次序就会改变。处理机响应各个中断源的中断服务请求和实际中断服务的先后次序如图 4.14 所示。

当处理机正在执行主程序时,中断源 D₁、D₂、D₃ 和 D₄ 同时请求中断服务,而且它们都没有被屏蔽。按照中断优先级的高低,处理机必然首先响应中断源 D₁ 的中断服务请求。处理机在处理中断源 D₁ 的过程中,通过硬件或软件改变中断源的中断屏蔽码,中断源 D₁ 被屏蔽,而中断源 D₂、D₃ 和 D₄ 没有被屏蔽。在实际开始执行 D₁ 的中断服务程序之前,要打开处理机的中断(开 CPU 中断)。这时,要从 D₂、D₃ 和 D₄ 三个没有被屏蔽的中断源中选择一个最高优

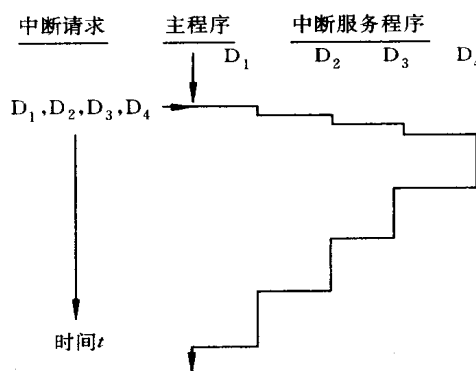


图 4.14 按照改变后的中断屏蔽码处理机响应中断源的中断服务请求和实际中断服务的过程

级的中断源 D₂,处理机响应中断源 D₂ 的中断请求。按照同样的过程,处理机在响应中断源 D₄ 的中断服务请求后,把所有中断源都屏蔽掉,因此,中断源 D₄ 的中断服务程序可以全部执行完成。在退出中断服务之前,中断源 D₄ 要恢复各个中断源的中断屏蔽状态,然后返回到被它中断的中断源 D₃ 的中断服务程序中。就这样逐级返回,直到返回主程序。

从图 4.14 中看到,处理机响应中断源的中断服务请求,其顺序是 D₁、D₂、D₃ 和 D₄,而实际中断服务的顺序为 D₄、D₃、D₂ 和 D₁,与中断源的中断优先级正好相反。因此,通过设置中断屏蔽码,可以通过软件,由程序员任意改变中断源的中断服务顺序,这正是我们所希望的中断系统的灵活性。

在为每个中断源或每级中断源设置一个中断屏蔽位的方法中,中断优先级实际上只在处理机响应中断源的中断服务请求时使用,即在所有请求中断服务的中断源中选择一个优先级最高的中断源首先为它服务。而处理机在执行中断源的中断服务程序时,并没有优先级的区分。

第二种、改变处理机优先级方法

中断优先级不仅在处理机响应中断源的中断服务请求时使用,而且为每个中断源的

中断服务程序也赋予同样的中断优先级。因为中断服务程序必须在处理机上执行,因此,把这种设置中断屏蔽的方法称为改变处理机优先级方法。

如果一台处理机共有 n 个中断源,则在处理机的状态字中需要设置 $\lfloor \log_2(n+1) \rfloor$ 个中断屏蔽位。这里,不把它称为中断屏蔽码,正好相反,称它为中断优先级。处理机本身的优先级一般设置为最低,例如,设置为“0级”,通常,处理机在运行主程序时,其优先级即为0级。另外,要为每一个中断源分别建立处理机状态字,通常把它们存放在主存储器的一个固定区域中。这些中断源的处理机状态字中同样也有一个中断优先级字段,而且每个中断优先级字段一般都可以由程序员通过软件来进行修改。

处理机在响应某一个中断源的中断服务请求后,就把属于这个中断源的处理机状态字作为当前处理机的状态字,这时候处理机的优先级也就有改变了,变成了程序员为这个中断源设置的中断优先级。这时,只有中断优先级高于当前处理机优先级的中断源才能中断当前的中断服务程序。

正常工作的情况下,在各个中断源的处理机状态字中设置的中断优先级应该与这个中断源本身的硬件中断优先级相同。这时,处理机响应中断源的中断服务请求和完成中断服务的过程将严格按照中断源的硬件中断优先级进行。

如果要改变中断源的中断服务顺序,即在有多个中断源同时请求中断服务时,让某些硬件中断优先级较低的中断源先得到处理机的服务,可以通过修改相关中断源的处理机状态字来实现。

下面,举一个简单的例子。

例如,某处理机共有4个中断源 D_1 、 D_2 、 D_3 和 D_4 ,它们在串行排队链中的硬件中断优先级从低到高分别为1级、2级、3级和4级。处理机本身的优先级最低,为0级。在中断源 D_1 、 D_2 、 D_3 、 D_4 的处理机状态字中,程序员为它们设置的优先级分别为4级、3级、2级、1级。

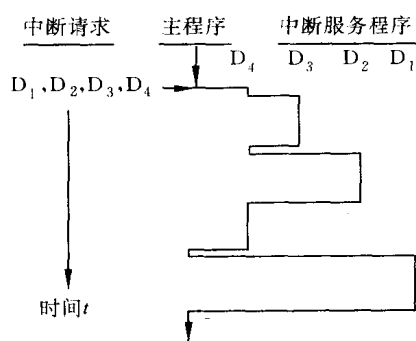


图 4.15 采用处理机优先级时,处理机响应中断源的中断服务请求和实际完成中断服务的过程

因为有4个中断源,因此在处理机状态字中要设置3个中断屏蔽位。其中,000为处理机本身的优先级,001~100分别表示4个中断源的中断优先级。如果当处理机正在执行主程序时,4个中断源同时请求中断服务,图4.15给出处理机实际响应中断源的中断服务请求和完成中断服务的过程。

当处理机运行主程序时,处理机的优先级最低,为0级。4个中断源同时请求中断服务,通过硬件排队器选择其中硬件中断优先级最高的中断源 D_4 ,处理机首先响应它的服务请求,并且把处理机的优先级改变为1级。这时,由于 D_1 、 D_2 、 D_3 三个中断源仍然在请求中断服务,而且,它们的硬件中断优先级都比当前处理机的优先级高,因此,处理机要响应其中硬件中断优先级最高的中断源 D_3 的中断请求,并且把处理机的优先级改变为2级。这时,还剩下 D_1 和 D_2 两个中断源在请求中断服务,它

们的硬件中断优先级都不高于当前处理机的优先级,因此,中断源 D_3 的中断服务程序能够一直执行完成。当处理机从中断源 D_3 的中断服务程序返回到中断源 D_4 的中断服务程序中时,处理机的中断优先级又变成了 1 级。这时,还没有得到处理机响应的两个中断源 D_1 和 D_2 中,只有中断源 D_2 的硬件中断优先级高于当前处理机的中断优先级。因此,处理机将立即响应中断源 D_2 的中断服务请求,又把处理机的优先级改变为 3 级。由于剩下的中断源 D_1 的硬件中断优先级低于当前处理机的优先级,处理机将把中断源 D_2 的中断服务程序全部执行完毕后会返回到中断源 D_4 的中断服务程序中。这时,最后一个没有被处理机响应的中断源 D_1 的硬件中断优先级与当前处理机的优先级相同,要等待处理机把中断源 D_4 的中断服务程序全部执行完成,并返回到主程序之后,中断源 D_1 的中断服务请求才能得到处理机的响应。由于中断源 D_1 是最后一个被处理机响应,并得到服务的中断源,因此,它的中断服务程序能够一直执行完成。在中断源 D_1 的中断服务程序全部执行完成,处理机返回来执行主程序时,全部中断源的中断服务请求也就处理完成了。

从图 4.15 中看到,虽然 4 个中断源 D_1 、 D_2 、 D_3 、 D_4 的硬件中断优先级是从低到高顺序排列的,并且 4 个中断源同时请求中断服务,但是,通过改变各个中断源所属的处理机状态字内的中断优先级,使得处理机实际完成中断服务的顺序改变成了 D_3 、 D_2 、 D_4 、 D_1 。当然,还可以根据需要,任意改变中断服务的顺序。

改变处理机优先级的方法与每一个中断源都设置一个中断屏蔽位的方法相比,两者都能由程序员通过软件来改变中断源的中断服务顺序,因此,它们都具有灵活性好的特点。它们的差别有两个,第一,两者使用的概念不同。前者使用的是中断屏蔽,后者使用的是中断优先级。第二,前者要为每一个中断源设置一个中断屏蔽位,所需要的位数比较多,而后者表示中断优先级所使用的位数要少得多。当然,前者使用起来要比后者方便些,例如,前者可以任意屏蔽掉一个或几个中断源,而后者只能屏蔽掉比某一个中断优先级低的中断源。从上面所举的两个例子中也可以看到它们的差别。

4.3 通道处理机

在大型计算机系统中,外围设备的台数一般比较多,设备的种类、工作方式和工作速度的差别也比较大。为了把对外围设备的管理工作从 CPU 中分离出来,从 IBM 360 系列机开始,普遍采用通道处理机技术。目前,在 IBM 370 系列机等几乎所有的 IBM 公司研制的计算机系统中都采用了通道处理机技术。在 IBM 公司研制的微型机和工作站等计算机系统中还采用了微通道技术。

4.3.1 通道的作用和功能

在大型计算机系统中,如果仅仅采用前面介绍过的程序控制、中断和 DMA 这三种基本的输入输出方式来管理外围设备,会引起如下两个问题:

1. 所有外围设备的输入输出工作全部都要由 CPU 来承担,CPU 的输入输出负担很重,不能专心于用户程序的计算。

低速外围设备,每传送一个字符都要由 CPU 通过执行一段程序来完成,而高速外围

设备虽然使用 DMA 方式减少了 CPU 的干预,但初始化等工作仍然需要 CPU 用程序来完成。在大型计算机系统中,这种输入输出工作对 CPU 的时间占用实际上是一种浪费。避免这种浪费的方法之一就是设置专门的输入输出处理机来分担全部或大部分的输入输出工作,例如,管理所有低速外围设备的输入输出工作,对 DMA 接口的初始化工作,控制 DMA 的数据传送,数据格式的变换,设备状态的检测等。这样,就能进一步提高整个计算机系统功能分散化的程度,充分发挥 CPU 的计算潜力。

2. 大型计算机系统中的外围设备台数虽然很多,但是一般并不同时工作。如果为每一台设备都配置一个接口,必然是一种浪费。特别是 DMA 接口,它的硬件代价很高。连接 DMA 接口的磁盘或磁带存储器等一般并不同时工作。

采用 DMA 方式传送数据,提高了输入输出数据的速度,节省了 CPU 的时间,但这是以对每一台快速外围设备都配备一个专用的 DMA 控制器作为代价的。在微型和小型计算机系统中,由于快速外围设备的台数很少,所使用的 DMA 控制器的数量有限。而在大型计算机系统中,快速外围设备的数量显著增加,就存在一个如何让 DMA 控制器能被多台设备共享的问题,以提高附加硬件的利用率。

为了使 CPU 摆脱繁重的输入输出负担和共享输入输出接口,在大型计算机系统中采用通道处理机是一种比较好的选择。

通道处理机能够负担外围设备的大部分输入输出工作,包括管理所有按字节传输方式工作的低速和中速外围设备,按数据块传输方式工作的高速外围设备,对 DMA 接口的初始化,设备故障的检测和处理等。通道处理机虽然不是一台具有完整指令系统的处理机,但是可以把它看作是一台能够执行有限输入输出指令,并且能够被多台外围设备共享的小型 DMA 专用处理机。

在一台大型计算机系统中可以有多个通道,一个通道可以连接多个设备控制器,而一个设备控制器又可以管理一台或多台外围设备,这样就形成了一个非常典型的输入输出系统的四级层次结构。

一般说来,通道的功能应该包括如下几个方面:

1. 接受 CPU 发来的输入输出指令,根据指令要求选择一台指定的外围设备与通道相连接。

2. 执行 CPU 为通道组织的通道程序,从主存中取出通道指令,对通道指令进行译码,并根据需要向被选中的设备控制器发出各种操作命令。

3. 给出外围设备的有关地址,即进行读/写操作的数据所在的位置。如,磁盘存储器的柱面号、磁头号、扇区号等。

4. 给出主存缓冲区的首地址,这个缓冲区用来暂时存放从外围设备上输入的数据,或者暂时存放将要输出到外围设备中去的数据。

5. 控制外围设备与主存缓冲区之间数据交换的个数,对交换的数据个数进行计数,并判断数据传送工作是否结束。

6. 指定传送工作结束时要进行的操作。例如,将外围设备的中断请求及通道的中断请求送往 CPU 等。

7. 检查外围设备的工作状态,是正常或故障。根据需要将设备的状态信息送往主存

指定单元保存。

8. 在数据传输过程中完成必要的格式变换,例如,把字拆卸为字节,或者把字节装配成字等。

为此,通道应该能够执行一组通道指令,而且还要具有完成上述功能的硬件。通道的主要硬件包括寄存器部分和控制部分,寄存器部分有:数据缓冲寄存器、主存地址计数器、传输字节数计数器、通道命令字寄存器、通道状态字寄存器。控制部分有:分时控制、地址分配、数据传送、数据装配和拆卸等控制逻辑。

通道对外围设备的控制通过输入输出接口和设备控制器进行,对于各种不同的外围设备,设备控制器的结构和功能也各不相同。然而,通道与设备控制器之间一般采用标准的输入输出接口来连接。通道命令通过标准接口送到设备控制器,设备控制器解释并执行这些通道命令,完成命令指定的操作,并且将各种外围设备产生的不同信号转变成标准接口和通道能够识别的信号。另外,设备控制器还能够记录外围设备的状态,并把状态信息送往通道和中央处理机。

4.3.2 通道的工作过程

在一般用户程序中,通过调用通道来完成一次数据输入输出的过程如图 4.16 所示,CPU 执行用户程序和管理程序,通道处理机执行通道程序的时间关系如图 4.17 所示,主要过程分为如下三步进行:

1. 在用户程序中使用访管指令进入管理程序,由 CPU 通过管理程序组织一个通道程序,并启动通道。

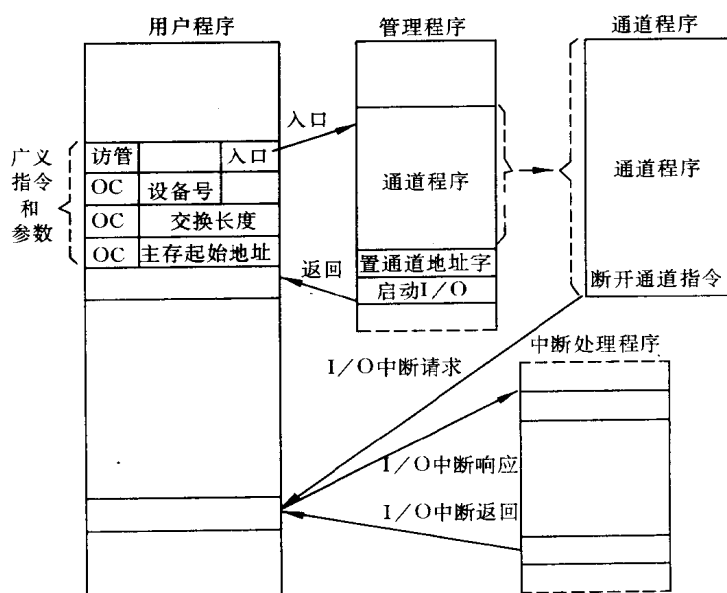


图 4.16 通道完成一次数据传输的主要过程

在多任务或多用户系统中,输入输出指令属于特权指令,一般用户程序不允许使用这

些指令。如果在用户程序中要进行输入输出操作,必须通过一条请求输入输出的广义指令进入操作系统,通过调用操作系统的管理程序来使用外围设备。

广义指令由一条访管指令和若干个参数组成,如图 4.16 所示。访管指令的地址码部分实际上就是这条访管指令要调用的管理程序入口地址。当用户程序执行到要求进行输入输出操作的访管指令时,产生自愿访管中断请求。CPU 响应这个中断请求后,转向管理程序入口。

管理程序根据广义指令提供的参数,如设备号、交换长度和主存起始地址等信息来编制通道程序。通道程序编制好后,放在主存储器中与这个通道相对应的通道程序缓冲区中。另外,在管理程序中还要把通道程序的入口地址置入主存储器的通道地址单元。在管理程序的最后,用一条启动输入输出设备指令来启动通道开始工作。

CPU 执行用户程序,用户程序调用管理程序及通道处理机执行通道程序的时间关系如图 4.17 所示。

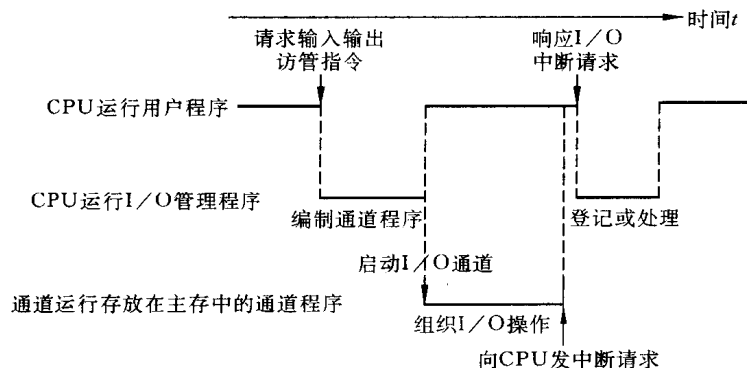


图 4.17 通道程序、管理程序和用户程序的执行时间关系

启动输入输出设备指令是一条主要的输入输出指令,它属于特权指令,在被访管指令调用的系统管理程序的最后一条执行。启动输入输出指令的工作流程如图 4.18 所示。先选择指定的通道和子通道,如果它们是在线而且是空闲的,就从主存中取出通道地址字,按照通道地址字给出的通道程序起始地址从主存的通道缓冲区中取出第一条通道指令。通道指令经过校验,如果指令格式是正确的,再选择指定的设备控制器和设备。如果被选择的设备是在线的,就向它发启动命令。设备被启动后,将向通道发回回答信号,如果设备的回答是一个全“0”字节,表示这台设备已经接受并执行了启动命令,设备的启动过程也就全部完成了。在上述过程中,如果任何一个地方不正确,表示启动输入输出设备没有成功,形成相应的条件码并结束启动过程。通道通过检测这些条件码,能够知道设备为什么没有启动成功。

2. 通道处理机执行 CPU 为它组织的通道程序,完成指定的数据输入输出工作。从图 4.17 中给出的时间关系可以看出,通道处理机执行通道程序是与 CPU 执行用户程序并行进行的。

通道被启动后,CPU 就可以退出操作系统的管理程序,返回到用户程序中继续执行

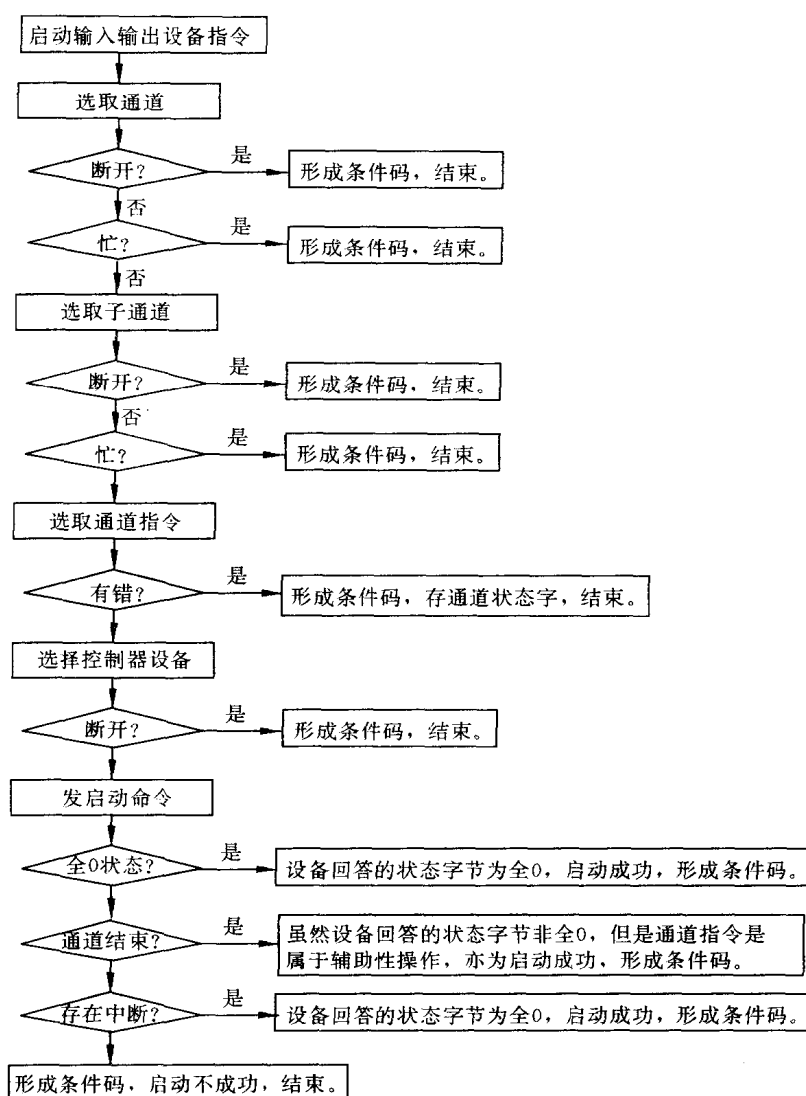


图 4.18 启动输入输出设备的工作过程

原来的程序,而通道开始与设备之间的数据传送。当通道处理机执行完通道程序的最后一条通道指令“断开通道指令”时,通道的数据传输工作就全部结束了。

3. 通道程序结束后向 CPU 发中断请求。CPU 响应这个中断请求后,第二次进入操作系统,调用管理程序对输入输出中断请求进行处理。

如果是正常结束,管理程序进行必要的登记等工作,如果是故障、错误等异常情况,则进行例外情况处理。然后,CPU 返回到用户程序继续执行。

这样,每完成一次输入输出工作,CPU 只需要两次调用管理程序,大大减少了对用户程序的打扰。当系统中由多个通道同时工作时,CPU 与多种不同类型、不同工作速度的外围设备可以充分并行工作。

在通道与设备之间的数据传送过程中,如果在同一个通道中有多台设备同时工作,则要反复重新选择设备,即找出当前要传送数据的是哪一台设备。对于低速设备,每传送完一字节就要重新选择一次设备,而对于高速设备,通常每传送完一个数据块重新选择一次设备。当然,如果一个通道只管理一台高速设备,那么,完成一次数据传送过程只需要做一次设备选择工作。

4.3.3 通道种类

根据多台外围设备共享通道的不同情况,可将通道分为三种类型:字节多路通道、选择通道和数组多路通道,这三种类型的通道与 CPU、设备控制器和外围设备的连接关系如图 4.19 所示。

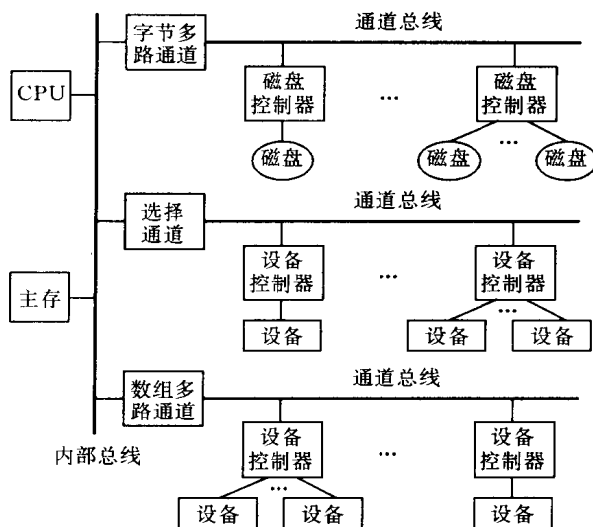


图 4.19 三种类型的通道与 CPU、设备控制器和外围设备的连接关系

4.3.3.1 字节多路通道

字节多路通道(byte multiplexor channel)是一种简单的共享通道,主要为多台低速或中速的外围设备服务。字节多路通道采用分时方式工作,依靠它与 CPU 之间的高速数据通路分时为多台设备服务。

字节多路通道可以有不同的工作方式。如果连接在通道上的各个设备轮流占用一个很短的时间片(通常小于 100 微秒)传输一个字节,或者说,不同的设备在它所得的时间片内与通道在逻辑上建立不同的传输连接,则称为字节交叉方式(byte-interleave mode)。如果允许一个设备一次占用通道比较长的时间传输一组数据,或者说,设备与通道的连接可以根据需要维持到一组数据全部传送完成,则称为成组方式(block mode)。两种工作方式之间的转换可以是自动进行的,它通过一个超时机制来进行控制。如果在超时机制预置的时间内,数据仍没有传送完毕,则自动转入成组方式工作,否则,继续采用字节交叉方式工作。

字节多路通道的组织结构如图 4.20 所示。它包含有多个子通道,每个子通道连接一个设备控制器。由于子通道的数目可以很多,例如,IBM 370 系列机的一个字节多路通道可以支持多达 256 个子通道,如果每个通道都有自己的一套硬件,必然造成总的设备量非常庞大,甚至达到不能容忍的程度。因此,目前的做法是:控制部分是公共的,由所有子通道共享,而寄存器部分每个子通道都有自己独立的一套。为了节省硬件,可以在主存储器中开辟出一个固定的区域来充当寄存器。对主存储器固定单元的访问速度完全能够满足低速和中速外围设备的要求。

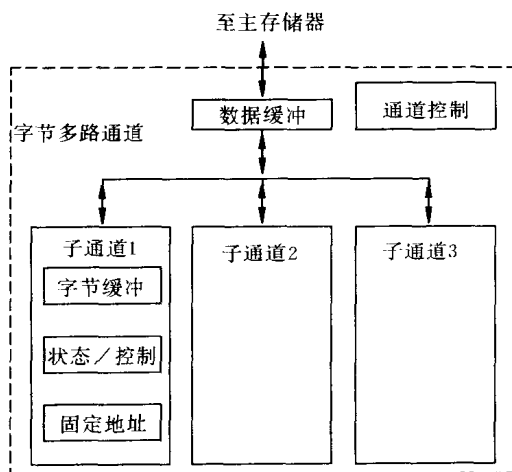


图 4.20 字节多路通道的结构

每个子通道最少需要有一个字节缓冲寄存器,一个状态/控制寄存器以及指明固定地址的少量硬件。与各个子通道有关的参数,如主存数据缓冲区地址、交换字节个数等都存放在主存固定单元中。当通道在逻辑上与某一设备连接时,就从主存相应的单元中把有关参数取出来,根据主存数据缓冲区地址访问主存储器,读出或写入一个字节,并将交换字节个数减 1,将主存数据缓冲区地址增量至下一个数据的地址。在这些工作都完成之后,就将通道与该设备在逻辑上断开。

4.3.3.2 选择通道

高速外围设备(如磁盘存储器等)需要很高的数据传输率,因此不能采用字节多路通道那样的控制方式。高速外围设备必须设置专门的通道在一段时间内单独为一台外围设备服务,但在不同的时间内仍可以选择不同的设备。一旦选中某一设备,通道就进入“忙”状态,直到该设备的数据传输工作全部结束为止,这就是选择通道(selector channel)。选择通道可以认为是只有一个以成组方式工作的子通道,只有一套完整的硬件,它逐个为物理上连接的几台高速外围设备服务。

选择通道的硬件组成如图 4.21 所示,它主要包含 5 个寄存器:数据缓冲寄存器、设备地址寄存器、主存地址计数器、交换字节数计数器、设备状态/控制寄存器等,另外,还有格式变换部件和通道控制部分等。

因为外围设备与通道控制器之间通常是以字节为单位传送数据的,而通道与主存储器之间要以字为单位传送数据,一个字的长度一般为 32 位或 64 位。数据格式变换部件完成字到字节的拆卸及字节到字的装配。

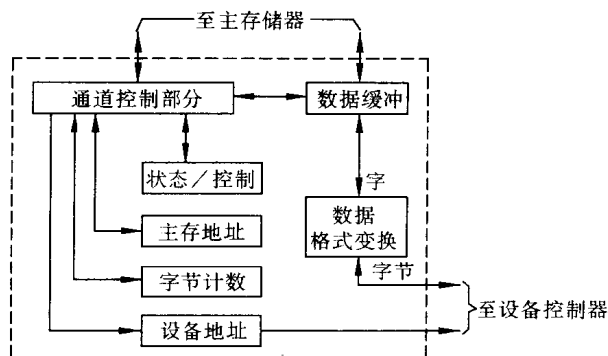


图 4.21 选择通道的结构

4.3.3.3 数组多路通道

把上面的字节多路通道和选择通道的特性结合起来就成为数组多路通道(block multiplexor channel)。它每次选择一个高速设备后传送一个数据块(对于磁盘和磁带等磁表面存储器,数据块大小通常为 512 个字节),并轮流为多台外围设备服务。因此,数组多路通道可以被看作是以成组方式工作的高速多路通道。

数组多路通道之所以能够并行地为多个高速外围设备服务,是因为这些高速外围设备并不能在整个数据输入输出时间内单独利用通道的全部传输能力。以从磁盘存储器读出一个文件的过程为例,可以分为如下三步:

第一步是定位,把读写磁头移动到记录该文件的磁道上,这要依靠机械动作来完成,称为定位时间,或找道时间,一般需要几十毫秒。

第二步是找扇区,等待读写磁头转动到记录该文件的起始扇区位置,称为找扇区时间,或等待时间。等待时间的长短主要与两个因素有关,一是磁盘的转速,二是磁头定位到所需要的磁道时,磁头所处位置与记录该文件的起始扇区位置的相对距离。因此,等待时间的长短是随机的,最长为磁盘转一周所需的时间,最短为零。取平均值,通常称为平均等待时间。目前,高速磁盘的转速已经达到每分钟 5 000 转以上,因此,磁盘存储器的平均等待时间一般小于 10 毫秒。

第三步是读出数据。目前,高速磁盘存储器的数据传输率已经达到每秒 33 兆字节以上,因此,读出一个扇区(512 个字节)只需要十几微秒时间。

通常把前两部分时间加起来称为磁盘存储器的寻址时间,第三部分时间称为数据传输时间。从上面的分析可以看出,磁盘存储器的寻址时间一般要比数据传输时间长两个数量级以上。因此,像选择通道那样,一个高速通道始终只为一台高速外围设备服务存在很大的浪费,并没有能够充分发挥高速通道的数据传输潜力,数组多路通道正是为了解决这一问题而提出来的。

数组多路通道在向一台高速设备发出定位命令后就立即从逻辑上与该设备断开,直到定位完成时再进行连接,发出找扇区命令后再一次断开,直到开始数据传送。因此,数组多路通道的实际工作方式是:通道在为一台高速设备传送数据时,有多台高速设备可以在定位或者在找扇区。

与选择通道相比,数组多路通道的数据传输率和通道的硬件利用率都很高,但是,由于在一次输入输出过程中要多次与同一台高速外围设备连接和断开,因此,增加了控制硬件的复杂性。

目前,大部分高性能计算机系统均采用数组多路通道,也有一些计算机系统采用选择通道,或这两种高速通道都采用。

4.3.4 通道中的数据传输过程

一个字节多路通道是分时为多台低速和中速外围设备服务的,在有 P 台设备同时连接到一个字节多路通道上时,它的数据传送过程如图 4.22 所示。

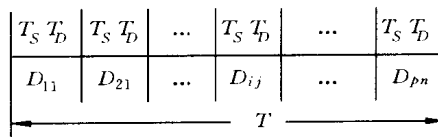


图 4.22 字节多路通道的数据传送过程

在图 4.22 中,每一个参数的含义如下,这些参数同样也适用于图 4.23 所示的选择通道和图 4.24 所示的数组多路通道。

T_s : 设备选择时间。从通道响应设备发出数据传送请求开始,到通道实际为这台设备传送数据所需要的时间。

T_d : 传送一个字节所用的时间,实际上就是通道执行一条通道指令,即数据传送指令所用的时间。

P : 在一个通道上连接的设备台数,且这些设备同时都在工作。

n : 每一个设备传送的字节个数,这里,假设每一台设备传送的字节数都相同,都是 n 个字节。

D_{ij} : 连接在通道上的第 i 台设备传送的第 j 个数据,其中: $i=1,2,\dots,p, j=1,2,\dots,n$ 。

T : 通道完成全部数据传送工作所需要的时间。

在字节多路通道中,通道每连接一个外围设备,只传送一个字节,然后又与另一台设备相连接,并传送一个字节,因此,在图 4.22 中,设备寻址时间 T_s 和数据传送时间 T_d 是间隔进行的。

当一个字节多路通道上连接有 P 台外围设备,每一台外围设备都传送 n 个字节时,总共所需要的时间 T 计算如下:

$$T_{\text{BYTE}} = (T_s + T_d) \cdot P \cdot n \quad (4.1)$$

选择通道在一段时间内只能单独为一台高速外围设备服务,当这台设备的数据传送

工作全部完成后,通道才能为另一台设备服务。因此,选择通道实际上是逐个为物理上连接的几台高速外围设备服务的。

选择通道的工作过程如图 4.23 所示,图中所用的参数与上面的字节多路通道相同,另外还有如下参数:

T_{Di} : 通道传送第 i 个数据所用的时间,其中: $i=1,2,\dots,n$ 。

D_i : 通道正在为第 i 台设备服务,其中: $i=1,2,\dots,p$ 。

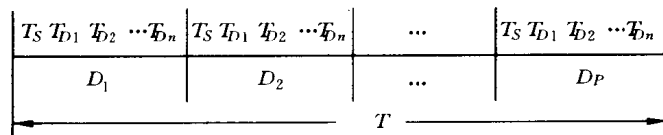


图 4.23 选择通道的数据传送过程

在选择通道中,通道每连接一个外围设备,就把这个设备的 n 个字节全部传送完成,然后再与另一台设备相连接,因此,在图 4.23 中,在一个设备寻址时间 T_s 之后,有连续 n 个数据传送时间 T_D 。

当一个选择通道上连接有 P 台外围设备,每一台外围设备都传送 n 个字节时,总共所需要的时间 T 计算如下:

$$T_{\text{SELETE}} = \left(\frac{T_s}{n} + T_D \right) \cdot P \cdot n \quad (4.2)$$

数组多路通道在一段时间内只能为一台高速设备传送数据,但同时,可以有多台高速设备在寻址,包括定位和找扇区。

数组多路通道的数据传送过程如图 4.24 所示,图中所用的参数与上面的字节多路通道和选择通道相同,另外还有如下一个参数:

k : 一个数据块中的字节个数。在一般情况下, $k < n$ 。对于磁盘、磁带等磁表面存储器, $k=512$ 。

数组多路通道每连接一台高速设备,一般传送一个数据块,传送完成后,又与另一台高速设备连接,再传送一个数据块,因此,在图 4.24 中,在一个设备寻址时间 T_s 之后,有连续 k 个数据传送时间 T_D 。

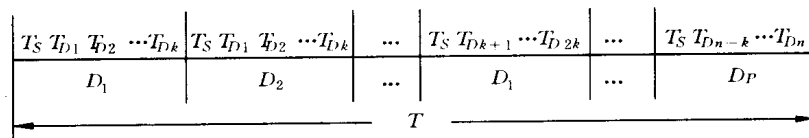


图 4.24 数组多路通道的数据传送过程

当一个选择通道上连接有 P 台外围设备,每一台外围设备都传送 n 个字节时,总共所需要的时间 T 计算如下:

$$T_{\text{BLOCK}} = \left(\frac{T_s}{k} + T_D \right) \cdot P \cdot n \quad (4.3)$$

4.3.5 通道的流量分析

通道流量又称为通道吞吐率,通道数据传输率等,它是指一个通道在数据传送期间,单位时间内能够传送的最大数据量,一般用字节个数来表示。一个通道在满负荷工作状态下的流量称为通道最大流量。通道最大流量主要与通道的工作方式(是指字节多路通道、选择通道和数组多路通道)、在数据传送期内通道选择一次设备所用的时间 T_s 和传送一个字节所用的时间 T_D 等因素有关。

根据通道流量的定义和上一节中的(4.1)、(4.2)和(4.3)式,可以得到三种通道的最大流量计算公式如下:

$$f_{\text{MAX-BYTE}} = \frac{p \cdot n}{(T_s + T_D)p \cdot n} = \frac{1}{T_s + T_D} \quad (4.4)$$

$$f_{\text{MAX-SELETE}} = \frac{p \cdot n}{\left(\frac{T_s}{n} + T_D\right)p \cdot n} = \frac{1}{\frac{T_s}{n} + T_D} \quad (4.5)$$

$$f_{\text{MAX-BLOCK}} = \frac{p \cdot n}{\left(\frac{T_s}{k} + T_D\right)p \cdot n} = \frac{1}{\frac{T_s}{k} + T_D} \quad (4.6)$$

根据字节多路通道的工作原理可知,它的实际流量是连接在这个通道上的所有设备的数据传输率之和,即:

$$f_{\text{BYTE}} = \sum_{i=1}^p f_i$$

对于选择通道和数组多路通道,在一段时间内,一个通道只能为一台设备传送数据,而且,这时的通道流量就等于这台设备的数据传输率。因此,这两种通道的实际流量就是连接在这个通道上的所有设备中数据流量最大的那一个:

$$f_{\text{SELETE}} = \max_{i=1}^p f_i$$

$$f_{\text{BLOCK}} = \max_{i=1}^p f_i$$

为了保证通道能够正常工作,即不丢失数据,各种通道实际流量应该不大于通道最大流量,即应该满足下列不等式关系:

$$f_{\text{BYTE}} \leq f_{\text{MAX-BYTE}}$$

$$f_{\text{SELETE}} \leq f_{\text{MAX-SELETE}}$$

$$f_{\text{BLOCK}} \leq f_{\text{MAX-BLOCK}}$$

两边的差值越小,通道的利用率就越高。当两边相等时,通道处于满负荷工作状态。在实际设计最大通道流量时,应留有一定的余量。例如,对于字节多路通道,通道的最大流量应略大于所有连接在这个通道上的设备的流量之和。如果一个字节多路通道的最大流量正好等于连接在这个通道上的所有设备的流量之和,当所有设备的数据传送请求集中出现时,有可能要丢失数据。这种情况可以从下面的例子中看到。

例 4.1 一个字节多路通道连接 D_1 、 D_2 、 D_3 、 D_4 、 D_5 共 5 台设备,这些设备分别每 10 微秒、30 微秒、30 微秒、50 微秒和 75 微秒向通道发出一次数据传送的服务请求,请回答

下列问题。

1. 计算这个字节多路通道的实际流量和工作周期。

2. 如果设计字节多路通道的最大流量正好等于通道实际流量,并假设对数据传输率高的设备,通道响应它的数据传送请求的优先级也高。5 台设备在 0 时刻同时向通道发出第一次传送数据的请求,并在以后的时间里按照各自的数据传输率连续工作。画出通道分时为各台设备服务的时间关系图,并计算这个字节多路通道处理完各台设备的第一次数据传送请求的时刻。

3. 从时间关系图上发现什么问题? 如何解决这个问题?

这个字节多路通道的实际流量为:

$$f_{\text{BYTE}} = \left(\frac{1}{10} + \frac{1}{30} + \frac{1}{30} + \frac{1}{50} + \frac{1}{75} \right) \text{MB/S} = 0.2 \text{MB/S}$$

通道的工作周期为:

$t = \frac{1}{f_{\text{BYTE}}} = 5 \text{ 微秒/字节}$, 包括通道选择设备的时间 T_s 和为设备传送一个字节所用的时间 T_D 。

5 台设备向通道请求传送数据和通道为它们服务的时间关系如图 4.25 所示,向上的箭头表示设备的数据传送请求,有阴影的长方形表示通道响应设备的请求并为设备服务所用的时间间隔,包括通道选择设备的时间和为设备传送一个字节所用的时间,这两部分的时间之和为 5 微秒。

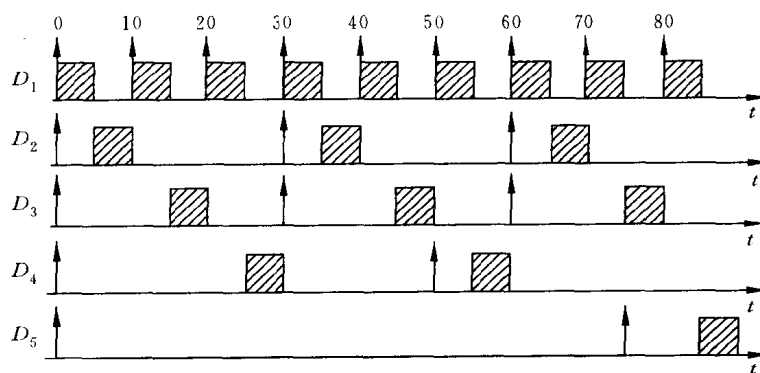


图 4.25 字节多路通道响应设备请求和为设备服务的时间关系图

在图 4.25 中,5 台设备在 0 时刻同时向字节多路通道发出第一次传送数据的请求,通道处理完各设备这个第一次请求的时间如下:

处理完设备 D_1 的第一次请求的时刻为 5 微秒;

处理完设备 D_2 的第一次请求的时刻为 10 微秒;

处理完设备 D_3 的第一次请求的时刻为 20 微秒;

处理完设备 D_4 的第一次请求的时刻为 30 微秒。

设备 D_5 的第一次请求没有得到通道的响应,直到第 85 微秒通道才开始响应设备 D_5 的服务请求,这时,设备已经发出了两个传送数据的服务请求,因此,第一次传送的数据有

可能要丢失。

当字节多路通道的最大流量与连接在这个通道上的所有设备的数据流量之和非常接近时,虽然能够从宏观上保证通道流量平衡,不会丢失数据,但是,由于传输速度高的设备频繁发出服务请求,并且优先得到通道的响应,某些低速设备可能在比较长一段时间内得不到通道的响应,如图 4.25 中得设备 D_5 。

只要仔细分析图 4.25 中时间关系不难发现,如果对所有设备的请求时间间隔取最小公倍数,那么,在这一段时间内通道的流量是平衡的,即所有设备的每一次服务请求都能够得到通道的响应,但是,在任意一台设备的任意两次时间传送请求之间并不能保证都能得到通道的响应。

根据上面的分析,为了保证字节多路通道能够正常工作,即不丢失数据,可以采取下列几种方法:

第一种方法,增加通道的最大流量,保证连接在通道上的所有设备的数据传送请求能够及时得到通道的响应。

第二种方法,动态改变设备的优先级。例如,在图 4.25 中,只要在 30 至 70 微秒之间临时提高设备 D_5 的优先级,那么,设备 D_5 的第一次数据传送请求就能及时得到通道的响应,其他设备的数据传送请求也能正常地得到通道的响应。当然,也可以采用临时降低设备优先级的办法,其效果是相同的。

第三种方法,增加一定数量的数据缓冲器,特别是对优先级比较低的设备。例如,在图 4.25 中,只要为设备 D_5 增加一个数据缓冲寄存器,那么,它的第一次数据传送请求可以在第 85 微秒处得到通道的响应,第二次数据传送请求可以在第 145 微秒处得到通道的响应。所有设备的数据都不会丢失。

4.4 输入输出处理机

采用输入输出处理机来分担中央处理机的输入输出任务是使用很普遍的一种方案。输入输出处理机又称外围处理机,I/O 处理机等,缩写为 IOP,或 PPU。输入输出处理机主要用在除 IBM 公司以外的其他计算机公司研制的巨型、大型计算机系统中,甚至在有些中小型及微型计算机系统中也有输入输出处理机。

4.4.1 输入输出处理机的作用

通道处理机实际上并不能看成是独立的处理机,因为它的指令(通道指令)系统很简单,只有面向外围设备的控制和数据传送的基本指令,而且没有大容量的存储器。在数据的输入输出过程中,通道处理机还需要由 CPU 来承担许多工作。在高性能的巨型和大型计算机系统中,如果仍然采用通道处理机方式,就会存在如下问题:

1. 每调用一次输入输出操作的前处理和后处理仍然要 CPU 来完成,需要两次用中断方式中断 CPU 的现行程序,调用操作系统的管理程序为输入输出操作服务。
2. 当外围设备或通道处理机出现异常情况时,通道处理机本身不能处理,要通过中断方式请求 CPU 来处理。

3. 对所传送数据的格式转换、码制转换、数据块整体的正确性检验等工作仍然要 CPU 来完成。

4. 文件的管理、设备的管理等操作系统的工作,通道处理机本身无能为力,需要 CPU 来实现。

由于上述原因,中央处理机资源往往得不到充分利用,造成很大的浪费。特别是在流水线计算机和向量计算机中,频繁的输入输出工作将使高性能的中央处理机无法充分发挥作用,运算速度严重下降。为此,采用输入输出处理机,使中央处理机进一步摆脱输入输出操作,使两种处理机并行工作,各负其责,各自充分发挥自己的作用,这是巨型、大型计算机系统,及一些输入输出任务比较繁重的计算机系统的必然选择。

图 4.26 是一种典型的采用输入输出处理机的计算机结构。一台输入输出处理机可以管理多台设备控制器,设备控制器与输入输出处理机之间采用标准的接口相连接。一台设备控制器可以管理多台同类型的输入输出设备。在一些高性能计算机中,CPU、I/O 处理机与主存之间要采用交叉开关网络来连接。

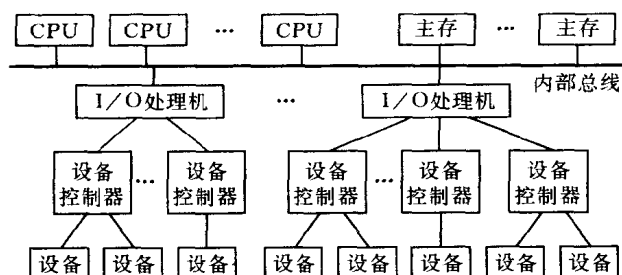


图 4.26 一种典型的采用输入输出处理机的计算机结构

输入输出处理机与通道处理机的主要差别是:输入输出处理机除了能够完成通道处理机的全部功能之外,还具有如下功能:

1. 码制转换。例如,ASCII 码与 EBCDIC 码之间的转换,ASCII 码与 BCD 码之间的转换,BCD 码与二进制数之间的转换等。

2. 数据校验和校正。由于外围设备的可靠性要比主机低,因此,在许多外围设备中都有相当复杂的校验和校正功能。在外围存储设备,如磁盘存储器和磁带存储器等中,一般采用硬件实现校验和校正,而在数据传输过程中的校验与校正一般要在输入输出处理机中通过程序来实现。

3. 故障处理。在通道处理机中,当外围设备或通道处理机本身出现故障时,要通过中断方式报告 CPU,由 CPU 来进行处理。而在输入输出处理机中,由于设置有算术逻辑指令和程序控制指令等,可以自己处理故障。

4. 文件管理。文件管理是一件相当烦琐的工作,输入输出处理机可以代替 CPU 完成文件管理的大部分工作。在有些计算机系统中,甚至专门设置有文件处理机。

5. 诊断和显示系统状态。通过定时运行诊断程序,可以诊断外围设备和输入输出处理机是否正常。

6. 处理人机对话。处理人机对话一般要通过反复调用管理程序来实现,有时把这种

管理程序称为监控程序。在设置有输入输出处理机的计算机系统中,一般把处理人机对话的管理程序放在输入输出处理机上执行。

7. 网络或远程终端可以直接连接到输入输出处理机上,由输入输出处理机完成远程用户服务工作。

除了以上工作之外,输入输出处理机还可以根据需要完成分配给它的其他任务,如数据库和知识库的管理工作等。

输入输出处理机之所以能够完成上述这些通道处理机所不能完成的工作,其中的关键是输入输出处理机除了具有数据的输入输出功能之外,还具有运算功能和程序控制等功能,即不仅能够执行输入输出指令,还能够执行算术逻辑指令和程序控制指令等,就像一个一般的处理机那样。

总之,输入输出处理机通常是一台独立的处理机,具有一定的运算功能,可以承担一般的外围处理机的输入输出、控制操作和运算处理等任务。另外,由于输入输出处理机具有自己的存储器,因此,不必通过主存储器就能完成与外围设备的数据交换,这样,可以进一步提高整个计算机系统的性能。

4.4.2 输入输出处理机的种类

输入输出处理机基本上是独立于中央处理机异步工作的,它可以与中央处理机共享主存储器,也可以有自己独立的存储器,不共享主存储器。每台输入输出处理机可以有自己独立的运算部件和指令控制部件,也可以由多个输入输出处理机共享同一个运算部件和指令控制部件。

根据是否共享主存储器,可以把输入输出处理机分为两大类:

1. 共享主存储器的输入输出处理机。许多早期的巨型和大型计算机系统一般采用这种方式。例如,CDC 公司的 CYBER,Texas 公司的巨型计算机 ASC,Burroughs 公司的 6700 等采用共享主存储器的连接方式。

每个输入输出处理机有一个小容量的局部存储器。输入输出处理机要执行的管理程序一般放在主存储器中为所有输入输出处理机共享,只当某一台输入输出处理机要用到时才通过加载或覆盖等方式把程序装入到它的局部存储器中。

2. 不共享主存储器的输入输出处理机。例如,早期的 STAT-100 巨型计算机。目前的大多数并行计算机系统都采用这种不共享主存储器的连接方式,各台输入输出处理机所运行的管理程序都存放在自己的大容量局部存储器中,因此,这种方式可以最大限度地减少对主存储器的压力。

根据运算部件和指令控制部件是否为各个输入输出处理机共享,也可以把输入输出处理机分为两类。

1. 合用同一个运算部件和指令控制部件的输入输出处理机。如 CDC-CYBER 和 ASC 等巨型计算机,并通过公用部件与主存储器相连接。这种输入输出处理机造价一般比较低,但控制相对比较复杂。

2. 独立运算部件和指令控制部件的输入输出处理机。例如,B-6700 大型计算机和 STAT-100 等巨型计算机。由于 VLSI 技术的高速发展,采用独立运算部件和指令控制部

件的输入输出处理机已经成为主流。而且,这种输入输出处理机往往都有各自的大容量存储器,具有更强的独立性。

根据各种计算机系统的具体情况和不同要求,输入输出处理机的结构有多种组织方式,例如:

1. 在有些计算机系统中,有多个输入输出处理机,而且从功能上进行分工,有的专门管理外围设备,有的专门管理文件系统,有的专门管理用户的人机会话工作,有的专门管理网络和远程终端,有的专门管理数据库或知识库等。

2. 在许多并行计算机和超级并行计算机系统中,以输入输出处理机作为主处理机,它除了担负全部输入输出任务之外,还运行操作系统,而由多个处理机或多个运算部件组成的并行处理系统仅作为运算的加速部件。

3. 在有的计算机系统中,用一台与中央处理机相同型号的处理机作为输入输出处理机,例如,有一种由两台 CRAY 大型计算机组成的系统,其中的一台计算机系统专门负责输入输出工作。

4. 随着集成电路技术的迅速发展,目前,许多计算机系统往往采用廉价的微处理器来专门承担输入输出任务,例如,Intel 公司的 8089 微处理器、80168 处理器等经常被用来作为专用的输入输出处理机。

4.4.3 输入输出处理机的特点

在高性能计算机系统中,为了能够使 CPU 摆脱繁重的输入输出任务,充分发挥高性能的 CPU 的运算功能,从 70 年代后期开始,首先在 CDC 公司研制的 6600 大型计算机系统中采用输入输出处理机方式。

下面以 CYBER170 巨型计算机的输入输出处理机为例来说明输入输出处理机的特点。CYBER170 输入输出处理机的结构如图 4.27 所示。

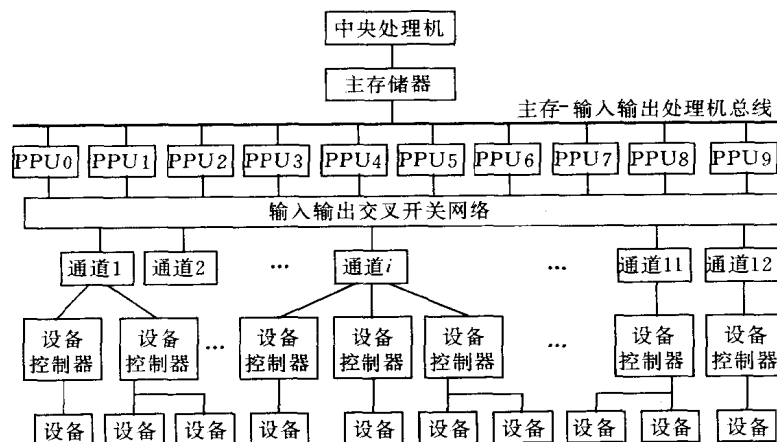


图 4.27 CYBER 1700 计算机系统的结构框图

10 台输入输出处理机 PPU0~PPU9 通过主存-输入输出处理机总线分时共享主存储器,并且通过输入输出交叉开关网络(I/O crossbar switching)共享 12 个输入输出通道。每个 PPU 有一个容量为 $4K \times 13$ 位(其中一位为奇偶校验位)的局部存储器。系统的监控程序常驻在 PPU0 的局部存储器中,控制台显示程序常驻在 PPU1 的局部存储器中。另外,各 PPU 的局部存储器中均装有自己的常驻程序。

中央处理机不能直接与外围设备打交道,当用户程序需要进行输入输出操作时,由中央处理机发出请求调用输入输出处理机,然后由输入输出处理机管理外围设备完成全部输入输出工作。

每台 PPU 都有相同的指令系统,共有 66 条指令,包括算术逻辑指令、访问存储器指令、输入输出指令及程序控制指令等。指令格式有 12 位的短指令和 24 位的长指令两种,如图 4.28 所示。OP 为操作码,D 为设备地址,M 为存储器地址。

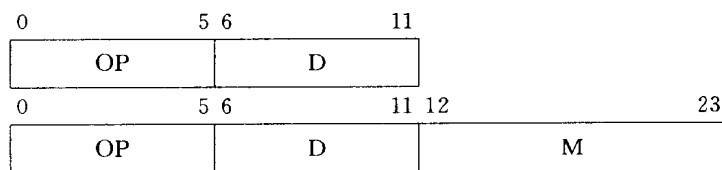


图 4.28 CYBER 1700 输入输出处理机的指令格式

每个 PPU 有 4 个寄存器,A 为累加寄存器,用于保存操作数、主存地址、输入输出字节数等,它也是 PPU 与其它部件之间进行通信的接口寄存器。P 寄存器为程序计数器。Q 寄存器为多功能寄存器,用于保存直接地址、间接地址、通道号或转移计数值等。K 寄存器为指令寄存器,用来保存指令操作码和指令执行周期。

图 4.27 中的通道与上一节中介绍的通道处理机有本质的不同,这里的通道仅仅是数据通路,或者理解为一种简单的接口。通道主要由一个双向的通道寄存器和两个触发器组成,其中一个为通道的“忙/闲”触发器,另一个是“满/空”触发器。当 PPU 要与外围设备交换数据时,先将设备号放在 A 累加器中,并将通道的“忙/闲”触发器和“满/空”触发器都置“1”,发出选择设备的信号。当设备收到该信号后,给 PPU 发一个回答信号,再将两个触发器清“0”,然后 PPU 把命令放在 A 累加器中,并启动设备。在数据传送期间,在 A 累加器中存放传输字节的个数。每当有一个数据通过通道时,就将“满/空”触发器置“1”,数据被取走后,“满/空”触发器清“0”,同时将 A 累加器中的字节个数减“1”。重复以上过程,直至 A 累加器中字节个数为“0”,整个数据传送过程全部结束,这时可以断开通道。

随着微处理机的迅速发展,微处理机的价格已经远远低于外围设备的价格,而且,微处理机具有一套完整的指令系统,能够分担中央处理机的许多工作,大大减轻了中央处理机的负担,提高了整个计算机系统的工作效率。因此,采用微处理机作为输入输出处理机,可以构成一个由中央处理机、输入输出处理机、主存储器、通道、设备控制器和各种外围设备相互独立的计算机系统,并且能够根据需要通程序动态地、灵活多变地控制它们之间的连接。

习 题 四

- 4.1 与计算机系统其他部分,如中央处理机、存储系统等相比,输入输出系统有什么特点?
- 4.2 常用的直接存储器访问(DMA)方式有哪几种?它们的工作原理和主要优缺点各是什么?
- 4.3 从一个中断源发出中断服务请求,到这个中断服务请求全部处理完成,程序返回到中断点所经过的过程称为中断处理过程。在一次完整的中断处理过程中,主要做哪些工作?其中,哪些必须用硬件实现?哪些必须用软件实现?哪些既可以用硬件实现也可以用软件实现?
- 4.4 有5个中断源 D_1 、 D_2 、 D_3 、 D_4 和 D_5 ,它们的中断优先级从高到低分别是1级、2级、3级、4级和5级。这些中断源的中断优先级、正常情况下的中断屏蔽码和改变后的中断屏蔽码见下表。每个中断源有5位中断屏蔽码,其中,“0”表示该中断源被屏蔽,“1”表示该中断源开放。

中断源名称	中断优先级	正常的中断屏蔽码					改变后的中断屏蔽码				
		D_1	D_2	D_3	D_4	D_5	D_1	D_2	D_3	D_4	D_5
D_1	1	1	1	1	1	1	1	0	0	0	0
D_2	2	0	1	1	1	1	0	1	0	0	0
D_3	3	0	0	1	1	1	1	0	1	0	0
D_4	4	0	0	0	1	1	1	1	0	1	1
D_5	5	0	0	0	0	1	1	1	1	0	1

- (1) 当使用正常的中断屏蔽码时,处理机响应各中断源的中断服务请求的先后次序是什么?实际的中断处理次序是什么?
- (2) 当使用改变后的中断屏蔽码时,处理机响应各中断源的中断服务请求的先后次序是什么?实际上中断处理的优先次序是什么?
- (3) 如果采用改变后的中断屏蔽码,当 D_1 、 D_2 、 D_3 、 D_4 和 D_5 这5个中断源同时请求中断服务时,画出处理机响应中断源的中断服务请求和实际运行中断服务程序过程的示意图。
- (4) 假设从处理机响应中断源的中断服务请求开始,到运行中断服务程序中第一次开中断所用的时间为1个单位时间,处理机运行中断服务程序的其他部分所用的时间为4个单位时间。当处理机在执行主程序时,中断源 D_3 、 D_4 和 D_5 同时发出中断服务请求,过3个单位时间之后,中断源 D_1 和 D_2 同时发出中断服务请求。采用改变后的中断屏蔽码,画出处理机响应各中断源的中断服务请求和实际运行中断服务程序过程的示意图。
- 4.5 某处理机共有4个中断源,分别为 D_1 、 D_2 、 D_3 和 D_4 。要求处理机响应中断源的中断

服务请求的次序从高到低分别为 D_1 、 D_2 、 D_3 、 D_4 ，而处理机实际为各中断源服务的先后次序为 D_3 、 D_2 、 D_4 、 D_1 。每个中断源有 4 位中断屏蔽码，其中，“0”表示该中断源被屏蔽，“1”表示该中断源开放。

(1) 请设计各中断源的中断优先级和中断屏蔽码。

(2) 如果处理机在运行主程序时，同时有 D_1 和 D_2 两个中断源请求中断服务，而在运行中断源 D_2 的中断服务程序的过程中，中断源 D_3 和 D_4 又同时请求中断服务，请画出处理机响应各中断源的中断服务请求和实际运行中断服务程序过程的示意图。

- 4.6 假设一台处理机共有 D_1 、 D_2 、 D_3 、 D_4 和 D_5 等 5 个中断源，它们的硬件中断优先级和在中断源的处理机状态字中由程序员设置的软件中断优先级见下表。处理机在运行主程序时，其中断优先级最低，为“0”级。

中断源名称	硬件中断优先级	软件中断优先级	中断响应优先次序	实际中断处理次序
D_1	7(最高)	4		
D_2	6	5		
D_3	5	7		
D_4	4	6		
D_5	3	5		

(1) 在处理机状态字中至少需要设置多少位中断屏蔽码？

(2) 在表中填写“中断响应优先次序”和“实际中断处理次序”。最先响应、最早得到处理的中断源填“1”，其余依次为 2、3、4、5。

(3) 当处理机正在执行主程序时，5 个中断源同时请求中断服务，画出处理机实际响应中断源的中断服务请求和运行中断服务程序过程的示意图。

- 4.7 一个字节多路通道连接有 5 台设备，它们的数据传输率如下表：

设备名称	D_1	D_2	D_3	D_4	D_5
数据传输速率(KB/s)	100	33.3	33.3	20	10
服务优先级	1(最高)	2	3	4	5

(1) 计算这个字节多路通道的实际工作流量。

(2) 为了使通道能够正常工作，请设计通道的最大流量和工作周期。

(3) 当这个字节多路通道工作在最大流量时，5 台设备都在 0 时刻同时向通道发出第一次传送数据的请求，并在以后的时间里按照各自的数据传输速率连续工作。画出通道分时为各台设备服务的时间关系图，并计算这个字节多路通道处理完各台设备的第一次数据服务请求的时刻。

- 4.8 一个字节多路通道连接有 4 台外围设备，每台设备发出输入输出服务请求的时间间

隔、它们的服务优先级和发出第一次服务请求的时刻如下表：

设备名称	DEV ₁	DEV ₂	DEV ₃	DEV ₄
发服务请求间隔	10 微秒	75 微秒	15 微秒	50 微秒
服务优先级	1(最高)	4	2	3
发出第一次请求时刻	0 微秒	70 微秒	10 微秒	20 微秒

- (1) 计算这个字节多路通道的实际流量和工作周期。
 - (2) 在数据传送期间,如果通道选择一次设备的时间为 3 微秒,传送一个字节的时间为 2 微秒,画出这个字节多路通道响应各设备请求和为设备服务的时间关系图。
 - (3) 从(2)的时间关系图中,计算通道处理完成各设备第一次服务请求的时刻。
 - (4) 从(2)画出的时间关系图中看,这个字节多路通道能否正常工作(不丢失数据)?为什么?
 - (5) 在设计一个字节多路通道的工作流量时,可以采用哪些措施来保证通道能够正常工作(不丢失数据)?
- 4.9 一台计算机系统有一个选择通道,两个数组多路通道,一个字节多路通道带有 3 个子通道。各通道的工作速度如下表：

通道名称		连接在这个通道上的设备的数据传送速率(KB/S)
字节多路通道	子通道 1	100, 50, 50, 25, 20, 5
	子通道 2	60, 60, 60, 45, 15, 10
	子通道 3	100, 100, 80, 80, 80, 60
数组多路通道 1		4 000, 4 000, 4 000, 3 000, 3 000
数组多路通道 2		4 000, 4 000, 4 000, 3 500, 3 000
选 择 通 道 1		5 000, 5 000, 5 000, 4 500, 4 000
选 择 通 道 2		6 000, 6 000, 5 000, 5 000, 5 000

- (1) 分别计算各通道和子通道的实际流量和工作周期。
- (2) 如果这台计算机系统的速度为 1GIPS,指令和数据的字长都是 32 位。指令 Cache 的命中率为 99%,数据 Cache 的命中率为 95%。假设平均每执行一条指令需要读或写一个操作数,这些操作数大部分来自通用寄存器,只有 20%来自存储系统。主存储器的字长为 32 位,请设计主存储器的访问周期和数据传输率。

第五章 标量处理机

只有标量数据表示和标量指令系统的处理机称为标量处理机。标量处理机是一种最通用,也是使用最普遍的处理机。

设计处理机的基本任务之一是要缩短解释指令的时间,即提高处理机指令执行的速度。通常提高指令执行速度的途径有如下三种:

1. 提高处理机的工作主频。在 50 年代和 60 年代主要采用这种技术,每 3 到 4 年,处理机执行指令的速度要提高一个数量级左右。目前,处理机的工作主频已经达到 500MHz 以上,提高处理机工作主频的速度已经明显减慢。

2. 采用更好的算法和设计更好的功能部件。例如,采用 RISC(精简指令系统计算机)技术减少执行指令的平均周期数;采用多位乘法 and 多位除法来缩短乘法和除法的执行时间;设计新型 ROM 乘法器等。

3. 多条指令并行执行,称为指令级并行技术。这是目前和将来提高处理机指令执行速度的一条主要途径。其中又有三种基本方法,第一种是采用流水线技术,称为流水线处理机或超流水线处理机(superpipelining)。第二种是在一个处理机中设置多个独立的功能部件,例如,在一个处理机中设置独立的定点算术逻辑部件、浮点加法部件、乘除法部件、访问存储器部件、分支操作部件等,称为多操作部件处理机或超标量处理机(superscalar)。也可以把超流水线技术与超标量技术结合起来,称为超标量超流水线处理机。第三种是超长指令字(very long instruction word, VLIW)技术,在一条指令中设置有多个独立的操作字段,每个字段可以分别独立地控制各个功能部件并行工作。目前,前两种技术已经相当成熟,已经研制出了多种高性能的超标量和超流水线处理机,而超长指令字技术还在进一步研究中。

本章以介绍流水线技术为主,包括先行控制技术、流水线原理、流水线性能分析、非线性流水线的调度方法、局部数据相关和全局数据相关的处理方法等,最后介绍超标量处理机和超流水线处理机等。

5.1 先行控制技术

先行控制(look-ahead)技术最早在 IBM 公司研制的 STRETCH 机器中采用。目前,许多处理机中都已经采用了这种技术,包括超流水线处理机和超标量处理机等。

先行控制技术的关键是缓冲技术和预处理技术,以及这两者的结合。通过对指令流和数据流的预处理和缓冲,能够尽量使指令分析器和指令执行部件独立地工作,并始终处于忙碌状态。

5.1.1 指令的重叠执行方式

一条指令的执行过程可以分为多个阶段,具体的分法要根据各种处理机的情况而确定。在图 5.1 中把一条指令的执行过程分为 3 个阶段。其中,取指令是按照指令计数器的

内容访问主存储器,取出一条指令送到指令寄存器。指令分析是指对指令的操作码进行译码,按照给定的寻址方式和地址字段中的内容形成操作数的地址,并用这个地址读取操作数,操作数可能主存储器中,也可能在寄存器中。指令执行是指根据操作码的要求,完成指令规定的功能,在此期间,要把运算结果写到寄存器或主存储器中。因此,在指令执行过程的三个阶段都可能要访问主存储器。另外,在指令分析或指令执行过程中还要完成指令计数器更新,为读取下一条指令作准备。

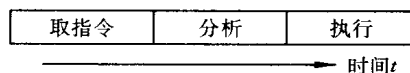


图 5.1 一条指令的执行过程

当有多条指令要在处理机中执行时,可以有多种执行方式:

1. 顺序执行方式。指令的执行过程如图 5.2(a)所示。采用顺序执行方式执行 n 条指令所用的时间为:

$$T = \sum_{i=1}^n (t_{\text{取指令}i} + t_{\text{分析}i} + t_{\text{执行}i}) \quad (5.1)$$

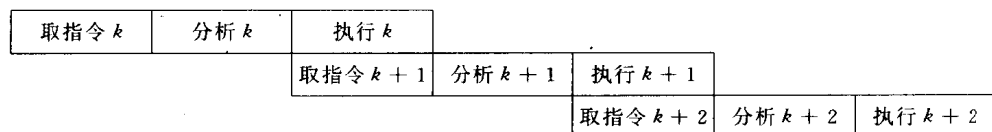
如果取指令、分析指令和执行指令的时间都相等,每段的时间都为 t ,则执行 n 条指令所用的时间为:

$$T = 3nt \quad (5.2)$$

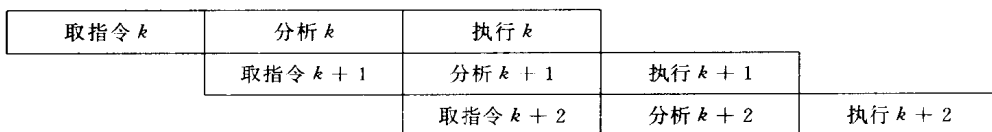
采用顺序执行方式的优点是控制简单,节省设备。主要缺点有两个,一是处理机执行指令的速度慢。只有当上一条指令全部执行完之后,下一条指令才能够开始执行,即在任何时刻,处理机中只有一条指令在执行。二是功能部件的利用率很低。例如,在取指令和分析指令时,主存储器是忙碌的,但指令执行部件是空闲的,同样,在执行指令时,指令执行部件是忙碌的,但主存储器和指令分析部件等经常是空闲的。因此,顺序执行方式并不能充分发挥各个功能部件的作用,实际上是一种浪费。



(a) 顺序执行方式



(b) 一次重叠执行方式



(c) 二次重叠执行方式

图 5.2 指令的几种执行方式

2. 一次重叠执行方式。这是一种最简单的重叠方式,如 5.2(b)图所示,把执行第 k 条指令与取第 $k+1$ 条指令同时进行。

如果执行一条指令的两个过程的时间均相等,则执行 n 条指令所用的时间为:

$$T = (1 + 2n)t \quad (5.3)$$

采用一次重叠执行方式的优点主要有两个,一是指令的执行时间缩短了近二分之一,二是功能部件的利用率明显提高。主存储器可以基本上处于忙碌状态,其他功能部件的利用率也得到提高。缺点是需要增加一些硬件,控制过程也要复杂一些。例如,为了能够在执行第 k 条指令的同时,分析第 $k+1$ 条指令,必须再增加一个指令寄存器。用原来的指令寄存器存放当前正在执行的第 k 条指令,而新增加的一个指令寄存器存放新取出来的第 $k+1$ 条指令。

3. 二次重叠执行方式。为了进一步提高指令的执行速度,可以把取第 $k+1$ 条指令提前到与分析第 k 条指令同时进行,同样,分析第 $k+1$ 条指令与执行第 k 条指令同时进行,如图 5.2(c)所示。

如果执行一条指令的三过程的时间相等,则执行 n 条指令所用的时间为:

$$T = (2 + n)t \quad (5.4)$$

采用二次重叠执行方式能够使指令的执行时间缩短近三分之二,这是一种理想的指令执行方式。在正常情况下,处理机中同时有三条指令在执行。

然而,为了能够实现这种理想的指令重叠执行方式,处理机的结构要作比较大的改变,必须采用先行控制方式。

5.1.2 先行控制方式的原理和结构

采用二次重叠执行方式,在处理机中同时有三条指令分别在取指令、分析和执行。要使指令能够正确地重叠执行,必须解决如下两个问题:

第一,为了实现取指令、分析指令和执行指令同时进行,需要有独立的取指令部件、指令分析部件和指令执行部件。因此,要把顺序执行方式中的一个集中的指令控制器,分解成三个相对独立的控制器,它们是:存储控制器,简称存控;指令控制器,简称指控;运算控制器,简称运控。

第二,要解决访问主存储器的冲突问题。例如,取指令时要访问主存储器,分析指令时可能要取操作数,执行指令时可能要向主存储器写运算结果。在一般机器中,指令和数据是混合存放在同一个主存储器中的;而且,在一个存储器周期中只能访问一个存储单元,这种常规的主存储器体系结构无法实现指令的重叠执行。通常,有以下三种方法可以解决这个问题:

1. 把主存储器分成两个独立编址的存储器,一个专门存放指令,称为指令存储器,简称指存,另一个专门存放操作数,称为数据存储器,简称数存。两个存储器可以同时独立访问;这样,就解决了取指令和读操作数的冲突。如果再规定,在执行指令阶段产生的运算结果只写到通用寄存器中,不写到主存,那么,取指令、分析指令和执行指令就可以同时进行。目前,有一些计算机系统就是这样做的。

在许多高性能处理机内部,一级 Cache 一般都设置有两个,其中一个是指令 Cache,

另一个是数据 Cache。这样,可以减少取指令和读操作数的访问冲突。这种结构被称为哈佛结构。

采用指令存储器和数据存储器分开的方法有一个明显缺点,即它对于汇编语言程序员和机器语言程序员是不透明的。

2. 指令和数据仍然混合存放在同一个主存储器内,采用第三章中介绍的低位交叉存取方式,在一个存储器周期中可以访问多个存储单元。如果处理机同时执行的取指令和读操作数所访问的不是同一个存储体,则可以实现指令重叠执行。如果正好访问同一个存储体,则指令无法重叠执行。当然,也采用高速 Cache,以缩短取指令和取操作数的时间。然而,这种方法不能从根本上解决访问存储器的冲突问题。

3. 解决访问存储器冲突的根本办法是采用先行控制技术。先行控制技术的关键是缓冲技术和预处理技术。缓冲技术是在工作速度不固定的两个功能部件之间设置缓冲栈,用以平滑它们的工作。在采用先行控制方式的处理机中,一般要设置四个缓冲栈。预处理技术是把进入运算器的指令都处理成寄存器-寄存器型(RR 型)指令,它与缓冲技术相结合,为进入运算器的指令准备好所需要的全部操作数。在采用了缓冲技术和预处理技术之后,运算器能够专心于数据的运算,从而大幅度提高指令的执行速度。因此,先行控制技术是现代计算机系统中被普遍采用的一项重要技术。

5.1.2.1 处理机结构

如图 5.3 所示,只要在处理机内部设置一定容量的指令缓冲栈,把指令分析器所需要的指令事先取到指令缓冲栈中,而不必访问主存储器。这样,就能够使取指令、分析指令和执行指令重叠起来执行。

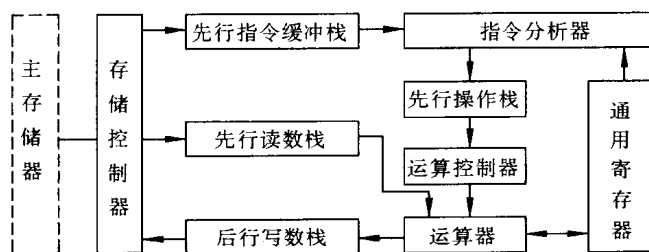


图 5.3 采用先行控制方式的处理机机构

指令缓冲栈又称为先行指令缓冲栈,这是因为它把后续的指令“先行”取出,从而为指令分析器分析新的指令作好准备,先行指令缓冲栈的结构如图 5.4 所示。它除了有一个指令缓冲寄存器堆之外,还有一套自己的控制逻辑,用来控制指令缓冲栈采用先进先出的方式工作,以保证指令的执行顺序不致混乱。

只要指令缓冲栈还没有全部充满,它就自动向存储控制器发出取指令的请求。同样,指令分析器每分析完一条指令也自动向指令缓冲栈发出取下一条指令的请求,指令取出以后就把先行指令缓冲栈中的指令作废。由于指令分析器取走指令的速度和从主存储器中取来指令的速度都是随机的,因此,指令缓冲栈中的指令数目是动态变化的。另外,在有先行指令缓冲栈的处理机中,要设置两个程序计数器,一个是先行程序计数器 PC_1 ,用来

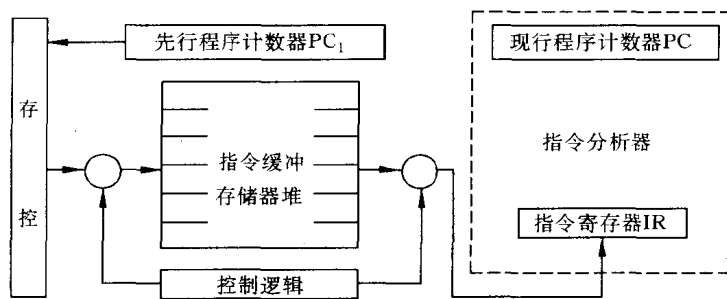


图 5.4 先行指令缓冲栈的组成

指示到主存储器中取指令，另一个是现行程序计数器，它也就是原来意义上的程序计数器 PC，用来记录指令分析器当前正在分析的指令地址。

在一般处理机中，存储控制器把先行指令缓冲栈的取指令安排在最低优先级，其余依次为读操作数和写结果，通常把输入输出请求安排在最高优先级。因此，先行指令缓冲栈是利用主存储器的空闲时间来取指令的。只要主存储器的频带宽度足够，就能够保证先行指令缓冲栈从主存储器中取到指令。

如果指令分析器每次取指令都能够在先行指令缓冲栈中得到，则取指令只需要很短的时间就能够完成，因此，可以把取指令与分析指令合并到一起，从而构成如图 5.5 所示的一次重叠执行方式。

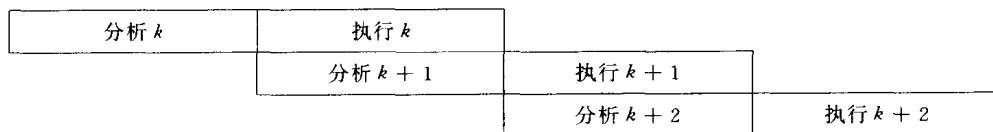


图 5.5 先行控制方式中的一次重叠执行

在采用先行控制方式之后，一次重叠执行方式是把一条指令的执行过程分为“分析”和“执行”两个子过程。从图 5.5 中可以看到，“执行 k ”与“分析 $k+1$ ”重叠进行。在一般的一次重叠执行方式中，即使“分析 $k+1$ ”比“执行 k ”提前结束，“执行 $k+1$ ”也不能开始，即“执行 $k+1$ ”不能与“执行 k ”重叠进行，必须等“执行 k ”完成之后才能开始“执行 $k+1$ ”。同样，当“执行 k ”比“分析 $k+1$ ”提前结束时，“分析 $k+2$ ”也不紧接在“执行 k ”之后与“分析 $k+1$ ”重叠进行。

对于严格的一次重叠执行方式，无论任何时刻，在指令分析部件和指令执行部件内都只有相邻的两条指令在重叠执行。因此，在处理机中只需要设置一套指令分析部件，即指令控制器，设置一套指令执行部件，即运算控制器和运算器。由于这种控制方式比较简单，在许多计算机中得到广泛应用。

如果分析指令和执行指令所需要的时间都是 t ，则采用一次重叠执行方式连续执行 n 条指令所需要的时间为：

$$T_{\text{重叠}} = (1 + n)t \quad (5.5)$$

实际上要达到这个速度还有许多困难，主要有如下三个问题需要解决：

1. 现代计算机的指令系统是很复杂的,各种类型的指令,其“分析”和“执行”所需要的时间相差往往很大。因此,指令分析部件和指令执行部件经常要相互等待,从而造成功能部件的浪费。

2. 如果“分析 $k+1$ ”所要读取的操作数正好是“执行 k ”的结果,则它们不能重叠执行,这种情况称为数据相关。另外还有控制相关和变址相关等。

3. 当出现转移或转子程序指令时,程序的执行过程就不是顺序的。这时,在先行指令缓冲器中预取的指令和已经分析完成的下一指令等都可能要作废。

在本节及以下的两节中,将分别介绍这三个问题的解决方法。采用这些方法,可以尽量减少速度的损失,充分发挥各个功能部件的作用。

5.1.2.2 指令执行时序

从图 5.6 中可以看出,在一般的一次重叠执行方式中,当“分析 $k+1$ ”完成之后,“执行 $k+1$ ”和“分析 $k+2$ ”都不能立即开始,必须等到“执行 k ”完成之后,“执行 $k+1$ ”和“分析 $k+2$ ”才能同时开始,这样,指令分析部件有一段时间是空闲的。同样,指令执行部件有时候也是空闲的。

在只有一套指令分析部件和一套指令执行部件的处理机中,为了充分发挥这两套功能部件的作用,尽量减少如图 5.6 中那样的速度损失,一种比较好的办法是采用缓冲技术。通过设置缓冲栈,两个工作时间长度不相等的功能部件可以各自完全独立地工作,以充分发挥每个功能部件的作用,这是计算机系统中经常采用的一项技术。

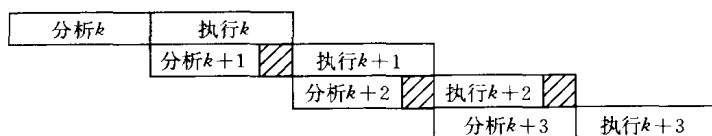


图 5.6 分析指令和执行指令时间不相等时的一次重叠执行方式

如图 5.3 所示,在采用先行控制方式的处理机中,除了要设置先行指令缓冲栈之外,还必须设置先行读数栈、先行操作栈和后行写数栈。在设置了这 4 个缓冲栈之后,存储控制器、指令分析器和运算控制器就能够各自完全独立地工作,指令的执行速度就会有比较大的提高。

在设置了先行缓冲栈后,分析指令和执行指令的时间关系如图 5.7 所示。“分析 k ”、“分析 $k+1$ ”、“分析 $k+2$ ”和“分析 $k+3$ ”是连续进行的。同样,“执行 k ”、“执行 $k+1$ ”、“执行 $k+2$ ”和“执行 $k+3$ ”也是连续进行的。指令分析部件和指令执行部件始终处于忙碌状态。

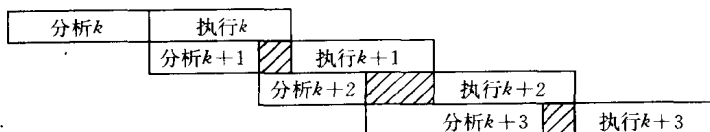


图 5.7 采用先行缓冲栈的指令执行过程

为了能够看得更清楚些,可以把图 5.7 改变成另一种表示方法,如图 5.8 所示。两个图的横坐标相同,都表示时间,图 5.7 的纵坐标表示指令,而图 5.8 的纵坐标表示功能部件。从图 5.8 中可以非常清楚地看到,指令分析部件在不间断地连续分析指令,而指令执行部件在不间断地执行指令。

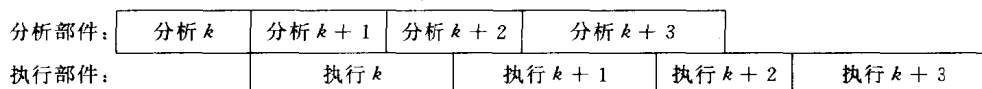


图 5.8 采用先行缓冲栈时指令执行过程的一种表示方法

采用先行控制方式,在理想情况下,指令执行部件应该一直是忙碌的,因此,处理机连续执行 n 条指令所需要的时间为:

$$T_{\text{先行}} = t_{\text{分析}1} + \sum_{i=1}^n t_{\text{执行}i} \approx \sum_{i=1}^n t_{\text{执行}i} \quad (5.6)$$

5.1.2.3 先行缓冲栈

在采用先行控制方式的处理机中,一般要设置四个先行缓冲栈,如图 5.3 所示,这四个缓冲栈的作用如下:

先行指令缓冲栈作为主存储器与指令分析器之间的一个缓冲部件,用于平滑主存储器和指令分析器的工作。当指令分析器分析某一条指令所需要的时间比较长,或者主存储器有空闲时,就从主存储器中多取出几条指令存放在先行指令缓冲栈中。而当指令分析器分析指令很快,或者主存储器比较忙时,指令分析器也能够从先行指令缓冲栈中得到所需要的指令。

指令分析器对已经存放在先行指令缓冲栈里的指令进行预处理,把处理机后的指令送入**先行操作栈**。对于无条件转移及条件转移等程序控制指令,一般在指令分析器中就能够直接执行,具体处理方法在下面一节中要专门介绍。各种运算型指令、移位指令、数据传送指令等都要先处理成寄存器-寄存器型(RR 型)指令,然后送入先行操作栈。例如,对于 RR 型指令,可以不作任何处理,直接送入先行操作栈。对于变址型(RX 型)和存储器型(RS 型)指令,指令分析器计算出主存有效地址后送入先行读数栈,由先行读数栈负责到主存储器中读操作数;同时用先行读数栈的寄存器编号替换原来指令中的主存地址码字段,形成 RR 型指令送入先行操作栈。为了与指令系统中原有的 RR 型指令相区别,通常把送入先行操作栈中的指令称为 RR * 型指令。对于立即数型(RI 型)指令,不必计算主存储器的有效地址,只要把指令中的立即数 I 送入先行读数栈,同样也用先行读数栈的寄存器编号替换原来指令中的立即数字段,形成 RR * 型指令送入先行操作栈。这样,经过指令分析器预处理之后送到先行操作栈中等待运算器执行的指令就都变成了统一格式的 RR * 型指令。执行这种指令所需要的操作数都存放在通用寄存器或先行读数栈中。

指令分析器每预处理完一条指令,就把形成的 RR * 型指令送入先行操作栈,使得指令分析器能够继续对后续的指令进行预处理。每当运算器执行完一条 RR * 型指令,就由

运算控制器从先行操作栈中取出下一条 $RR *$ 型指令。因此,先行操作栈是指令分析器和运算控制器之间的一个缓冲存储器;有了这个缓冲存储器,指令分析器和运算器就能够各自完全独立地工作。因为相对于正在运算器中执行的指令来讲,存放在先行操作栈中的 $RR *$ 型指令是后续指令“先行”预处理的结果,因此叫做先行操作栈。先行操作栈采用先进先出方式工作,它由一个先行指令寄存器堆和有关控制逻辑等组成。

先行读数栈由一组缓冲寄存器和有关控制逻辑等组成。每一个缓冲寄存器由三部分组成,包括先行地址缓冲寄存器、先行操作数缓冲寄存器和标志字段。也可以把先行地址缓冲寄存器和先行操作数缓冲寄存器合用一个寄存器。每当先行地址缓冲寄存器收到从指令分析器中送来的有效地址时,就把地址的有效标志置位,并开始向主存储器申请读操作数。从主存储器读出的操作数可以存放在先行读数栈的操作数缓冲寄存器中,也可以覆盖掉原来的先行地址寄存器中的地址,同时要置位数据有效标志。先行读数栈一般也采用先进先出方式工作。运算器能够直接从先行读数栈中取得所需要的操作数,而不必等待访问主存储器;因此,先行读数栈是主存储器与运算器之间的一种缓冲存储器,用来平滑运算器与主存储器的工作。因为先行读数栈中的操作数对于正在执行的指令而言,是把后续指令要用到的操作数“先行”取出,所以叫做先行读数栈。设置了先行读数栈后,在运算器中执行的指令,原来要访问主存储器的,现在变成了访问先行读数栈中的先行操作数缓冲寄存器。这样,指令的执行速度就能够大大加快。

如果在指令分析器中遇到向主存储器的“写数”指令,则把形成的有效地址送入后行写数栈的后行地址缓冲寄存器中,并把预处理好的 $RR *$ 型指令送入先行操作栈。当然,这条 $RR *$ 型指令中的目标寄存器就是后行写数栈的缓冲寄存器编号。当运算器执行这条 $RR *$ 型写数指令时,只要把要写到主存中去的数据送到后行写数栈的后行数据缓冲寄存器中即可,由后行写数栈负责把数据写回到主存储器。这样,运算器不必等待数据写回到主存,就可以继续执行后续的指令。

与先行读数栈类似,**后行写数栈**也由一组缓冲寄存器和有关控制逻辑等组成。每一个缓冲寄存器必须包括后行地址缓冲寄存器、后行数据缓冲寄存器和标志字段三个部分,其中的后行地址缓冲寄存器和后行数据缓冲寄存器不能象先行读数栈那样合用。每当接收到从运算器送来的结果数据时,就把相应的数据有效标志置位,控制逻辑自动向主存储器发出写数请求。当写数操作完成之后,也要置位有关标志。因为后行写数栈中将要写回主存储器去的数据,相对于正在运算器执行的指令来讲,是其先前指令的“后行”数据,因此称作为后行写数栈。后行写数栈是主存储器与运算器之间的一个缓冲存储器。设置了后行写数栈,运算器中的指令执行速度就可以加快。

从上面的分析中可以看到,先行控制技术实质上是缓冲技术和预处理技术相结合的结果。通过对指令流和数据流的预处理和缓冲,能够尽量使指令分析器和指令执行部件独立地工作,并始终处于忙碌状态。在采用先行控制方式的处理机中,一个程序的执行情况如图 5.9 所示。

从图 5.9 中可以看出,先行控制方式与一次重叠执行方式有着根本的不同。先行控制方式在指令分析器和指令执行部件中同时处理的是两条可能不相邻的指令。在处理机内

部,同时可能有多条指令在重叠执行。显然,它比一次重叠执行方式的并行性更高。

指令地址	指令执行情况
... $k - i - 1$	已经执行完成的指令
$k - i$... $k - 1$	在“后行写数栈”中等待把结果写到主存储器中的指令
k	正在“指令执行部件”中执行的指令
$k + 1$... $k + j$	已经由“指令分析器”预处理完成,被存放在“先行操作栈”中的RR*型指令,指令所需要的操作数也已经读到“先行读数栈”中
$k + j + 1$... $k + j + n$	指令已经由“指令分析器”预处理完成,存放在“先行操作栈”中,指令所需要的操作数还没有读到“先行读数栈”中
$k + j + n + 1$	正在“指令分析器”中进行分析 and 预处理的指令
$k + j + n + 2$... $k + j + n + m$	已经从主存储器中预取到“先行指令缓冲栈”中的指令
$k + j + n + m + 1$...	还没有进入处理机的指令

图 5.9 采用先行控制方式时一个程序的执行情况

通常把先行控制方式中的先行指令缓冲栈、先行读数栈、先行操作栈和后行写数栈统称为先行控制器。先行控制器与指令分析器一起构成先行控制方式中的指令控制部件。而原来的运算器及运算控制器一起组成指令执行部件。

5.1.2.4 缓冲深度的设计

在设计先行控制器时,比较困难的一件事情是各个缓冲栈中的缓冲寄存器个数设置多少个,即所谓“缓冲深度”问题。如果缓冲寄存器的个数设置得太少,往往起不到缓冲的作用,指令分析器和指令执行部件不能连续地工作。相反,如果缓冲寄存器个数设置得太多,不仅浪费设备,而且控制逻辑也复杂。如何设计先行控制器中四个缓冲栈的深度是先行控制器设计中必须要解决的一个关键性问题。

在各个缓冲栈的入端和出端,数据(或指令)流动的速度是动态变化的,要建立一个准确的数学模型非常困难。因此,一般采用静态分析方法,再通过系统模拟来确定各个缓冲栈的缓冲深度。

所谓静态分析是通过分析两种极端情况来计算缓冲深度。下面以设计先行指令缓冲栈为例来说明缓冲深度的静态分析方法。

一种极端情况是:先行指令缓冲栈已经完全充满,假设缓冲深度为 D_1 ;这时,在先行指令缓冲栈的输出端,指令流出的速度最快;而在先行指令缓冲栈的输入端,指令流入的速度最慢,即指令分析器连续分析的都是最容易分析的指令,每条指令所需要的分析时间很短。假设这种指令序列的最大长度为 L_1 ,平均分析一条这种指令的时间为 t_1 ;而到主存储器中取指令,每次都需要很长的时间,假设平均取一条这种指令的时间为 t_2 。另外,在先

行指令缓冲栈从完全充满到全部被取空的过程中,假设指令分析器所分析的指令条数为 L_1 条,则从主存储器中取到先行指令缓冲栈中的指令条数是 $L_1 - D_1$ 条。很明显应该满足如下关系:

$$L_1 t_1 = (L_1 - D_1) t_2 \quad (5.7)$$

计算出缓冲深度为:

$$D_1 = \left\lceil \frac{L_1 \cdot (t_2 - t_1)}{t_2} \right\rceil \quad (5.8)$$

如果这种指令流的连续长度超过 L_1 ,则先行指令缓冲栈将被取空,指令分析器就没有指令可分析,被迫处于等待状态。

另一种极端情况是:先行指令缓冲栈原来是空的,没有存放任何指令,并假设它的缓冲深度为 D_1 ;这时,先行指令缓冲栈的输入端,指令流入的速度最快,而输出端指令流出的速度最慢,即到主存储器中取指令,每次所需要的时间都是最短的;假设这种指令序列的最大长度为 L_2 ,平均取一条这种指令的时间为 t_2' ;而指令分析器连续分析的都是最难分析的指令,每条指令所需要的分析时间很长,假设平均分析一条这种指令的时间为 t_1' 。另外,从先行指令缓冲栈完全是空的到全部充满指令的过程中,假设从主存储器中取到先行指令缓冲栈中的指令条数为 L_2 条,则指令分析器所分析的指令条数是 $L_2 - D_1$ 条。很明显应该满足如下关系:

$$(L_2 - D_1) t_1' = L_2 t_2'$$

计算出缓冲深度为:

$$D_1 = \left\lceil \frac{L_2 \cdot (t_1' - t_2')}{t_1'} \right\rceil \quad (5.9)$$

如果这种指令流的连续长度超过 L_2 ,则先行指令缓冲栈被全部充满,它也就失去了缓冲的作用。

由于从指令分析器出来的指令是要送到指令执行部件执行的,因此,指令最终从处理机流出的速度要由指令在执行部件中的执行速度来决定。在一般程序中,执行时间短的指令所占的比例要远远大于执行时间长的指令。执行时间短的指令一般有加减法指令、逻辑运算指令、移位指令等,执行时间长的指令主要是乘法。另外,还有一些“吸收型”指令在指令分析器中就直接执行完成,不需要送到运算器中执行,如转移指令等。因此,指令分析器分析指令的速度要大于运算器执行指令的速度。鉴于这些原因,一般用前一种方法来计算缓冲深度。下面,举一例子。

一个采用先行控制方式的处理机,指令分析器分析一条指令用一个周期,到主存储器中取一条指令装入先行指令缓冲栈平均要用 4 个周期。如果这种指令的平均长度 $L_1 = 9$,即 90% 的指令是执行时间短的指令。则根据(5.8)式,计算出先行指令缓冲栈的缓冲深度为:

$$D_1 = \left\lceil \frac{L_1 \cdot (t_2 - t_1)}{t_2} \right\rceil = \left\lceil \frac{9 \cdot (4 - 1)}{4} \right\rceil = 7 \quad (5.10)$$

图 5.10 表示这种情况下的先行指令缓冲栈的工作时间关系。

在第 1 个工作周期,指令分析器从先行指令缓冲栈中取走指令 $k + 1$;由于先行指令

缓冲栈中只剩下 6 条指令,缓冲栈未满,要向主存发出请求取指令到先行指令缓冲栈的信号。在第 4 个工作周期的末尾,指令 $k+8$ 从主存储器取到先行指令缓冲栈中;同时,先行指令缓冲栈再次向主存储器发请求取指令的信号。在第 8 个工作周期,指令分析器分析指令 $k+8$,这时先行指令缓冲栈中已经没有指令,但正好在这个周期的末尾,指令 $k+9$ 从主存储器取到先行指令缓冲栈中,在图中用 $1*$ 表示。在第 9 个工作周期,指令分析器分析指令 $k+9$,先行指令缓冲栈中再也没有任何指令。到第 10 个周期,指令分析器没有指令可分析,只好等待。

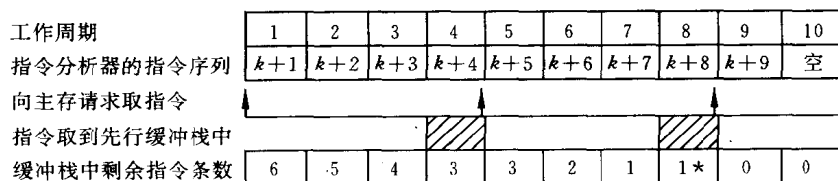


图 5.10 先行指令缓冲栈的工作时间关系图

从关系式(5.8)及图 5.10 中可以看出,提高主存储器的访问速度,如采用多体并行和低位交叉存取存储器,或者增加 Cache 的容量或级数,对减少缓冲深度很有效。另外,在实际确定缓冲深度时,还要选择一些典型程序,在现有的机器上,通过模拟的方法选取比较合理的缓冲深度。

其余 3 个缓冲栈的缓冲深度也可以采用类似的方法确定。在同一台采用先行控制方式的处理机中,各个缓冲栈的缓冲深度一般有如下关系:

$$D_1 \geq D_C \geq D_R \geq D_W \quad (5.11)$$

其中, D_1 是先行指令缓冲栈的缓冲深度, D_C 是先行操作栈的缓冲深度, D_R 是先行读数栈的缓冲深度,而 D_W 是后行写数栈的缓冲深度。

例如,IBM 370/165 机, $D_1 = 4, D_C = 3, D_R = 2, D_W = 1$ 。我国研制的两台大型计算机,其中一台, $D_1 = 8, D_C = D_R = 4, D_W = 2$ 。而另外一台, $D_1 = 12, D_C = D_R = 6, D_W = 2$ 。

5.1.3 数据相关

所谓相关(correlation)是指在一段程序的相近指令之间有某种关系,这种关系可能影响指令的重叠执行。通常,把相关分为两大类,一类是数据相关,另一类是控制相关。在执行本条指令的过程中,如果用到的指令、操作数、变址偏移量等正好是前面指令的执行结果,则必须等待前面的指令执行完成,并把结果写到主存或通用寄存器中之后,本条指令才能开始执行,这种相关称为数据相关。控制相关是指由条件分支指令、转子程序指令、中断等引起的相关。

本节主要介绍数据相关,下一节专门介绍控制相关。

在采用先行控制方式的处理机中,数据相关主要有四种。分别是指令相关、主存操作数相关、通用寄存器相关和变址相关等。

解决数据相关的方法通常有两种,一种方法是推后分析法,在遇到数据相关时,推后本条指令的分析,直至所需要的数据写入到相关的存储单元中。另一种方法是设置专用路

径,即不必等待所需要的数据写入到相关存储单元中,而是经专门设置的数据通路读取所需要的数据。

5.1.3.1 指令相关

看下面的两条指令:

```
k:      STORE    R1, k + 1
k + 1:  ...
```

其中,存在关系:

$$\text{结果地址}(k) = \text{指令地址}(k + 1) \quad (5.12)$$

第 $k + 1$ 条指令本身的内容取决于第 k 条指令的执行结果,则产生指令相关。

如果采用一次重叠执行方式,当“执行 k ”还没有完成时,取出的第 $k + 1$ 条指令显然是错误的。

在采用先行控制方式的处理机中,相关的情况还要严重得多。当执行部件正在执行第 k 条指令时,已经有多条指令经指令分析器预处理完成,存放在先行操作栈中;有一条指令正在指令分析器中进行预处理,可能还有更多的指令已经预取到先行指令缓冲栈中。如图 5.9 中,如果正在执行部件中执行的第 k 条指令要修改从第 $k + 1$ 到第 $k + j + n + m$ 这些指令中的任意一条指令,都可能造成程序执行结果发生错误。

要判断是否发生了指令相关,要把每一条指令的结果地址与先行操作栈中、指令分析器中和先行指令缓冲栈中的所有指令地址进行比较。如果发现相关,则在修改主存相关单元的同时,也要修改先行操作栈、或指令分析器、或先行指令缓冲栈中的相关指令。更为严重的是,有一些“吸收型”指令,如转移指令等,可能已经在指令分析器执行完成,不再进入先行操作栈,因此也无法再对它们进行修改了。

解决指令相关的根本办法是在程序设计中不允许修改指令。当然,不允许修改指令还有更重要的原因,即现代程序设计方法要求程序具有再入性,可以被递归调用等。另外,在程序执行过程中不修改指令也有利于程序的调试和诊断。

在 IBM 370 机中,有一条“执行”指令能够解决指令相关,又允许在程序执行过程中修改指令。当然,使用了“执行”指令的程序同样具有再入性,可以被递归调用。这条“执行”指令的格式如下:

0	7 8	11 12	15 16	19 20	31
EX(执行)	R ₁	X ₂	B ₂	D ₂	

“执行”指令本身并不实际执行,它执行的是由第二地址 $((X_2) + (B_2) + D_2)$ 决定的主存单元中的指令。这个主存单元一般不在指令区,而是在数据区。因此,即使这个主存单元被修改了,被修改的也只是数据,而不是指令。在程序执行过程中,可以先修改这条位于数据区中的指令,然后再执行“执行”指令。这样,实际上就达到了修改指令的目的。另外,“执行”指令在执行数据区中的指令时,还要用第一地址 (R_1) 指定的通用寄存器中的最后 8 位(第 24 到 31 位)与位于数据区内的,将要被实际执行的指令的第 8 至 15 位相“或”。由于这 8 位正

好是指令的关键地址部分,因此,这样做可以进一步提高“执行”指令的灵活性。

5.1.3.2 主存操作数相关

当指令的执行结果写到主存储器,所读取的操作数也取自主存储器时,就有可能发生主存操作数相关。看下面的两条三地址指令:

k : OP A_1, A_2, A_3 ; $A_1 = (A_2) \text{ OP } (A_3)$
 $k+1$: OP A_1, A_2, A_3 ; $A_1 = (A_2) \text{ OP } (A_3)$

如果发生:

$$\text{结果地址}(k) = \text{主存操作数地址}(k+1) \quad (5.13)$$

则发生主存操作数相关。

对于上面的两条指令,如果 A_1, A_2, A_3 都是主存的地址单元,当出现

$$A_1(k) = A_2(k+1) \quad (5.14)$$

或

$$A_1(k) = A_3(k+1) \quad (5.15)$$

时,就发生主存操作数相关。

由于在大多数处理机中,运算结果一般都写到通用寄存器,而不写到主存,因此,主存操作数相关出现的概率比较小。

解决主存操作数相关一般采用推后处理法。如图 5.11 所示,第 k 条指令的“结果写主存 A_1 单元请求”发生在一个周期的接近某末尾处,而第 $k+1$ 条指令的“读主存 A_1 单元请求”出现在同一个周期的开始位置。在设置有存储控制器的处理机中,只要把“写结果”的优先级安排得高于“读操作数”的优先级;由于存储控制器响应访问请求是定时进行的,它在一个周期的最末尾处对这一个周期中的所有访问源进行排队。

一般处理机中,对访问主存储器的请求,排队的优先级从高到低分别是输入输出请求、写结果、读操作数和取指令。从图 5.11 中可以看出,“写主存 A_1 单元请求”必然先得到响应,而“读主存 A_1 单元请求”由于没有及时得到存储控制器的响应,只得推后处理。因此,主存操作数相关也就自然就解决了。

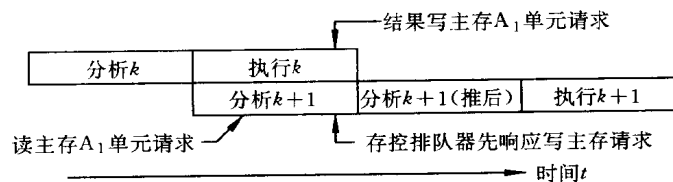


图 5.11 主存操作数相关及其处理方法

在设置有先行操作栈的处理机中,正在指令分析器中分析的指令与已经在指令执行部件中执行完成,需要进入后行写数栈中向主存写回运算结果的指令之间可能要相隔很多条指令。已经进入先行操作栈的任何一条指令在向主存请求读操作数时都可能与正在执行部件中执行的指令、正在后行写数栈中等待写结果到主存中的指令、甚至还在先行操作栈中的指令之间发生主存操作数相关。

解决这种操作数相关的办法是:对于刚进入先行操作栈中的指令在向主存读操作数之前,首先要把访问主存的地址与后行写数栈中的所有主存地址进行比较,如果发现有相等的,则先行操作栈的读操作数要暂缓进行,等到发生操作数相关的指令执行完成,并把结果写回到主存之后再开始读操作数。

5.1.3.3 通用寄存器数据相关

在寄存器-寄存器型(RR 型)指令和寄存器-存储器型(RS 型)指令的执行过程中可能发生通用寄存器数据相关。看下面两条指令:

k : OP R_1 , A_2 ; $R_1 = (R_1) \text{ OP } (A_2)$

$k+1$: OP R_1 , R_2 ; $R_1 = (R_1) \text{ OP } (R_2)$

如果发生:

$$R_1(k) = R_1(k+1) \quad (5.16)$$

称为 R_1 数据相关。如果发生:

$$R_1(k) = R_2(k+1) \quad (5.17)$$

称为 R_2 数据相关。

发生通用寄存器数据相关的情况与寄存器本身的结构和所采用的控制时序等也有关系。解决通用寄存器数据相关的方法有如下几种:

方法一、如果通用寄存器是用D型触发器构成的,而且在通用寄存器到运算器之间

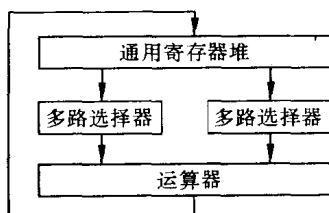


图 5.12 一种典型的运算器结构

建立有直接数据通路,即不设置缓冲寄存器或锁存器,则不会发生通用寄存器数据相关。如图 5.12 所示,因为 D 型触发器允许在同一节拍中实现寄存器之间的循环传送。

这种情况下,实际上不需要分析周期,在一个节拍中就可以完成从通用寄存器中通过两个多路选择分别读两个操作数到运算器,操作数在运算器完成运算,并把运算结果再写回到通用寄存器中。

如果构成通用寄存器的不是 D 型触发器,或者在通用寄存器到运算器之间设置有缓冲寄存器或锁存器,则可能要发生通用寄存器数据相关。下面,以 IBM 370 系列机中的通用寄存器为例来说明通用寄存器数据相关的解决方法。

方法二、分析指令推后一个周期执行。如图 5.13 所示,分析周期和执行周期都是 4 个节拍,在“分析 $k+1$ ”的最后两个节拍分别把两个操作数送入运算器的锁存器中,而“执行 k ”要到最后一个节拍才能把运算结果写到通用寄存器中。这时,当出现 (5.16) 或 (5.17) 式的关系时,就发生通用寄存器数据相关。

当发生通用寄存器数据相关时,为了保证送入通用寄存器中的数据是正确的,可以把“分析 $k+1$ ”推后一个周期到“执行 k ”完成之后再开始,实际上是分析指令和执行指令完全串行进行。

这种方法实现起来比较简单,通用寄存器堆只需要一个输出端口,运算器完成一次运

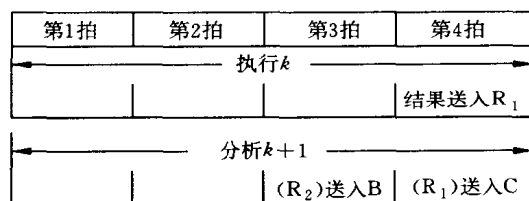


图 5.13 IBM 370 机的指令分析与执行的时间关系

算所需要的两个操作数分两个节拍从寄存器堆中读出。但它有一个明显的缺点是运算速度的损失比较大。

方法三、分析指令仅推后一个节拍。从图 5.13 中可以看出,“分析 $k+1$ ”可以少推后一些,不必推后一个周期。如图 5.14 所示,实际上,“分析 $k+1$ ”可以仅推后一个节拍。具体方法是:当判别到“分析 $k+1$ ”与“执行 k ”有数据相关时,在分析周期的第 3 个节拍首先把一个不相关的数据从通用寄存器堆中读出送到运算器的锁存器中,然后等“执行 k ”把数据写入通用寄存器堆之后,在“分析 $k+1$ ”的最后一个节拍把发生数据相关的那个操作数从通用寄存器堆中读出,送到运算器的另一个锁存器中。

方法四、设置专用数据通路。由于在运算型指令中把运算结果写到通用寄存器中的指令很多,占运算型指令的大多数,发生通用寄存器数据相关的概率相当高,因此,在许多处理机中为了减少因为通用寄存器数据相关造成的速度损失,专门用硬件设置一条专用路径来解决这种数据相关。

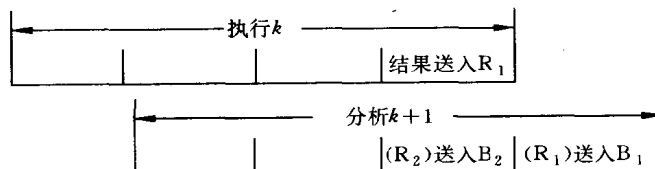


图 5.14 分析指令仅推后一个节拍的时间关系

如图 5.15 所示,在运算器的输出端到锁存器的输入端之间建立一条专用的数据通路。当出现(5.16)式或(5.17)式的关系时,“执行 k ”的运算结果在送入通用寄存器堆的同时,还要通过这条专用的数据通路送回到运算器的锁存器中;当然,这时要封锁通用寄存器到锁存器的数据通路。这样,无论是出现 $R1$ 数据相关还是 $R2$ 数据相关,只要在“执行 k ”的末尾,把发生数据相关的操作数送到运算器的锁存器中,“分析 $k+1$ ”就不必再推后,可以与“执行 k ”同时进行,就像没有发生数据相关时一样,指令分析器与指令执行部件仍然可以不间断地并行工作。

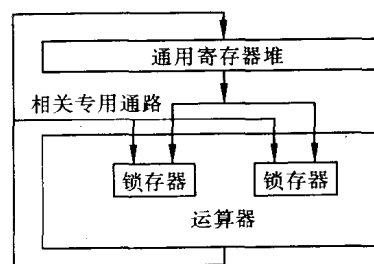


图 5.15 设置专用数据通路解决通用寄存器数据相关

实际上,设置相关专用通路是通过增加硬件为代价来解决通用寄存器数据相关的。它

的运算速度高,指令分析与执行可以完全重叠进行。而前两种时间推后方法则基本上不增加硬件,是以降低运算速度为代价来解决通用寄存器数据相关的。在实际设计处理机时,要权衡运算速度和硬件价格等因素,选择适当的方法。

5.1.3.4 变址相关

在许多处理机中,把通用寄存器兼作变址寄存器使用。由于在变址寄存器中存放的变址量在指令分析过程中要用它来计算有效地址,因此,与通用寄存器的数据相关类似,有可能发生变址相关。

因为计算有效地址是在指令分析的一开始进行的,因此,变址相关造成的后果要比通用寄存器数据相关更为严重。看下面三条指令:

k : OP R_1 , R_2 ; $R_1 = (R_1) \text{ OP } (R_2)$
 $k+1$: OP R_1 , $A_2(X_2)$; $R_1 = (R_1) \text{ OP } ((A_2) + (X_2))$
 $k+2$: OP R_1 , $A_2(X_2)$; $R_1 = (R_1) \text{ OP } ((A_2) + (X_2))$

由于 R_1 、 R_2 和 X_2 都是通用寄存器,如果发生:

$$R_1(k) = X_2(k+1) \quad (5.18)$$

称为一次变址相关,如果发生:

$$R_1(k) = X_2(k+2) \quad (5.19)$$

称为二次变址相关。

发生一次变址相关的原因与通用寄存器数据相关相同。发生二次变址相关的原因是:由于“执行 k ”的运算结果是在最末尾写入通用寄存器的,通常,寄存器写入数据之后需要一段稳定时间,然后才能再次读出送到指令分析部件中,假设这段时间为 Δt_1 。在“分析 $k+1$ ”及“分析 $k+2$ ”一开始,只经过一段很短的指令译码时间 Δt_2 之后,就要使用通用寄存器中变址量计算有效地址。因此,如果 $\Delta t_1 > \Delta t_2$,且满足关系式(5.18),则“分析 $k+2$ ”计算出来的有效地址将是错误的。

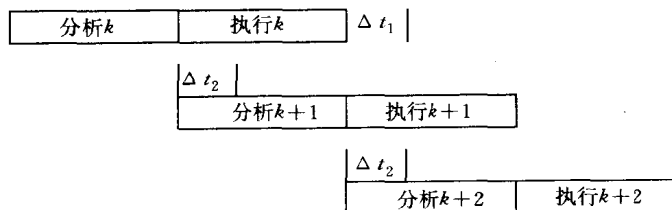


图 5.16 一次变址相关和二次变址相关的时间关系

解决变址相关的方法与解决通用寄存器数据相关类似,也可以采用推后分析和设置专用通路两种方法。如果采用推后分析法,对于二次变址相关,需要把“分析 $k+2$ ”推后一个周期或一个节拍。对于一次变址相关,要把“分析 $k+1$ ”推后两个周期或一个周期再加一个节拍。

由于变址相关出现的概率比较高,因此,在许多处理机中采用设置“变址相关专用通路”的办法来解决变址相关。如图 5.17 所示,在发生二次变址相关时,可以把“执行 k ”得

到的运算结果在送入通用寄存器堆的同时,也经过“变址相关专用通路”直接送到地址加法器中。这样,对于二次变址相关,可以不推后“分析 $k+2$ ”。同样,在设置了“变址相关专用通路”之后,对于一次变址相关,只要把“分析 $k+1$ ”推后一个周期到“执行 k ”完成之后进行即可。

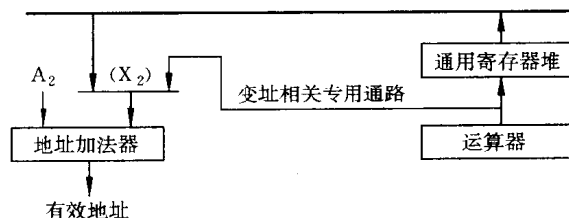


图 5.17 用变址相关专用通路解决变址相关

从上面的分析中可以看到,对于各种数据相关,主要有三种解决方法。第一种方法是采用硬件或软件的办法尽量避免数据相关发生。例如,对于指令相关,要求程序设计人员不采用修改指令的方法编写程序,或者在指令系统中增加一条“执行”指令来解决。对于主存操作数相关,通过安排存储控制器的访问优先级来解决。为了避免通用寄存器数据相关,只要采用 D 型触发器构成通用寄存器堆,并且通用寄存器堆在一个节拍中能够读出两个操作数并写回一个结果。第二种方法是在确保指令正确执行的前提下,推后指令分析。这实际上是一种为了节省设备而牺牲运算速度的解决办法。第三种方法是设置相关专用通路来解决数据相关,这种方法实际上是通过增加硬件设备来换取提高指令执行速度的一种办法。

5.1.4 控制相关

控制相关是指因为程序的执行方向可能被改变而引起的相关。可能改变程序执行方向的指令通常有无条件转移、一般条件转移、复合条件转移、子程序调用、中断等。本节主要介绍因各种转移指令引起的相关,对于子程序调用和中断引起的相关将在流水线一节中再介绍。

有些资料上把上一节中介绍的数据相关称为局部相关,而把本节将要介绍的控制相关称为全局相关。这是因为数据相关影响到的仅仅是本条指令附近的少数几条指令,而控制相关影响的范围要大得多,它可能引起程序执行方向的改变。

无条件转移指令和一般条件转移指令可以在指令分析器就执行完成,不需要送入先行操作栈和指令执行部件。因此,通常把这种指令称为“吸收型”指令。下面,分别介绍几种转移指令在先行控制器中的处理方法。

5.1.4.1 无条件转移

无条件转移指令在程序中的使用情况一般如下:

```

k:      ...
k+1:    JMP L
  
```

...: ...
L: ...

无条件转移指令一般能够在指令分析器中就执行完成,形成转移地址 L 同时送入先行程序计数器 PC_1 和现行程序计数器 PC 中,指令缓冲栈按照 PC_1 的指示重新开始向存储控制器申请取指令。

如图 5.18 所示,如果转移距离比较远,指令 L 不在先行指令缓冲栈中,则要将先行指令缓冲栈中的所有指令全部作废,指令分析器要等待一个“取指令 L”周期之后才能开始“分析 L”。如果转移的距离比较近,有可能指令 L 已经被取到了指令缓冲栈中。这时,只要作废先行指令缓冲栈中的部分指令,即作废从 $k+2$ 到 $L-1$ 之间的所有指令。指令分析器在完成“分析 $k+1$ ”之后,可以接着进行“分析 L”,即指令分析器仍然可以不停顿地连续工作。

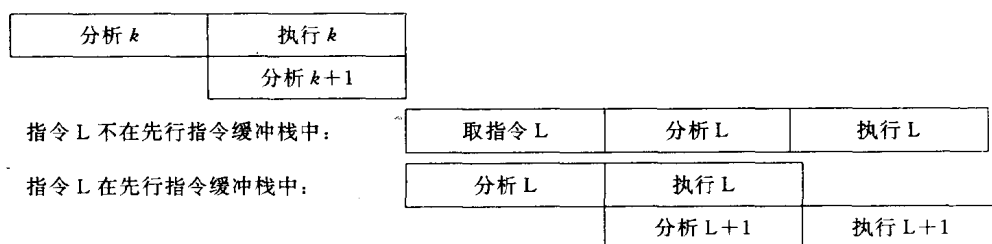


图 5.18 无条件转移指令的执行时序

在设置有先行操作栈的处理机中,当指令分析器“分析 $k+1$ ”时,由于在先行操作栈中可能还缓存有没有执行的指令,因此,无条件转移指令一般对指令执行部件的工作不会造成影响。

在有的处理机中,为了进一步减少无条件转移指令造成的影响,在先行指令缓冲栈的入口处增设一个专门处理无条件转移指令的指令分析器。当这个指令分析器分析到无条件转移指令时,就把转移地址 L 送入先行程序计数器 PC_1 中。这样,取到先行指令缓冲栈中的指令就都是有用的,不会再发生作废先行缓冲栈中指令的情况。采用这种方法能够减少处理机与主存储器之间的通信量,而且,无条件转移指令对后面的指令分析器和指令执行部件等将不再造成影响,就像程序中没有无条件转移指令一样。

在采用先行控制方式的处理机中,处理机执行程序的速度主要是由指令执行部件执行指令的速度决定的,这一点可以从上面的(5.6)关系式中看出。因此,无条件转移指令对程序执行速度的影响很小。

5.1.4.2 一般条件转移

在采用先行控制方式的处理机中,条件转移指令对程序执行速度造成的影响很大。先行控制部件的缓冲深度越深,造成的速度损失也可能越大。然而,可以利用大量的先行缓冲部件,采取一些新的措施来减少处理机的速度损失。

条件转移指令有两种,即一般条件转移指令和复合条件转移指令。其中,一般条件转移指令的转移条件来自上一条指令,或更前面的指令。而复合型条件转移指令直接根据本

条指令的执行结果决定是否转移。一般条件转移指令的执行时间关系下:

```

k:      ...           ; 置条件码 CC
k+1:    JMP(CC)  L    ; 如果 CC 为真转向 L, 否则继续执行 k+2
k+2:    ...
...:    ...
L:      ...

```

对于一般条件转移指令,相关最严重的情况发生在条件码是由上一条指令产生的。如图 5.19 所示,条件码要在“执行 k ”的末尾才能形成,而“分析 $k+1$ ”在一开始就要根据这个条件码判断转移条件是否成立。由于存在有先行操作栈,当指令分析器分析到条件转移指令时,必须停下来等待所需要的条件码。直到指令执行部件把先行操作栈中的全部指令都执行完成,在“执行 k ”结束时才能形成条件码。指令分析器根据这个条件码才能判断转移是否成功。

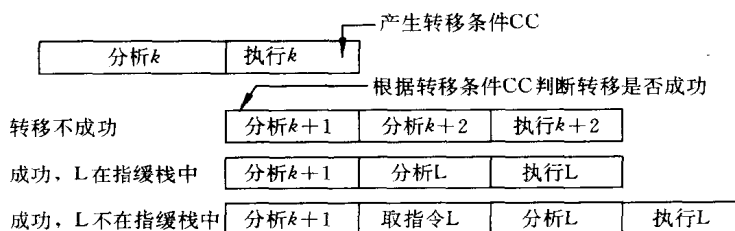


图 5.19 一般条件转移指令的执行时序

无论转移是否成功,第 $k+1$ 条转移指令都在指令分析器就已经执行完成,不再送往指令执行部件,因此,一般转移指令被称为“吸收型”指令。

如果转移不成功,尽管指令分析器要停顿一段时间,但是,处理机执行指令的速度损失不大。因为指令分析器在分析完条件转移指令后,可以继续“分析 $k+2$ ”,指令执行部件在“执行 k ”之后,只要等待一个周期,就又可以继续“执行 $k+2$ ”。在先行指令缓冲栈中预取的指令也都仍然有效。

如果转移成功,而且转移的距离比较近,有可能指令 L 已经被取到了先行指令缓冲栈中。这时,只要作废先行指令缓冲栈中从 $k+2$ 到 $L-1$ 之间的所有指令。指令分析器在完成“分析 $k+1$ ”之后,可以接着进行“分析 L”。如果转移距离比较远,指令 L 不在先行指令缓冲栈中,则要将先行指令缓冲栈的所有指令全部作废,指令分析器还要再等待至少一个“取指令 L”周期之后才能开始“分析 L”。在这种情况下,程序的执行过程与串行执行完全一样。

从上面的分析中可以看到,转移不成功对先行控制器的影响不大,而当转移成功时,不仅指令执行过程变成了完全串行,而且要作废已经取到先行指令缓冲栈中的大量指令,从而白白地增加了处理机与主存之间的通信量。因此,在采用先行控制方式的处理机中,要通过软件与硬件等多种手段来尽可能地降低转移成功的概率和减少因为转移成功对先行控制器造成的影响。

5.1.4.3 复合条件转移

复合型条件转移指令本身就是一条运算指令,并且直接根据本条指令的运算结果决定是否转移。例如,在第二章中介绍的 IBM 370 计算机中的计数转移,大于转移,减“1”为“0”转移等。复合型条件转移指令的执行时间关系下:

k: OP L ; 先执行 OP 操作,根据执行结果产生的条件码决定是否转向 L
k+1: ...
...: ...
L: ...

复合型条件转移指令不是吸收型指令,它必须送入先行操作栈中象其他运算型指令一样经指令执行部件执行,执行时序如图 5.20 所示。

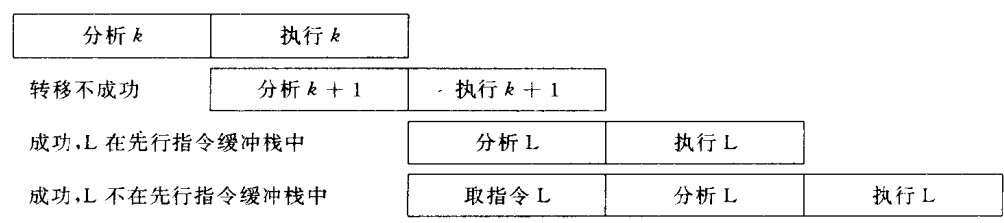


图 5.20 复合型条件转移指令的执行时序

如果转移不成功,只要采取适当的措施,复合型条件转移指令对先行控制器不造成任何影响,指令分析器和指令执行部件仍然继续工作,先行指令缓冲栈和先行操作栈中的指令仍然有效。这时,复合型条件转移指令就像一条普通的运算型指令一样。

如果转移成功,复合型条件转移指令对先行控制器造成的影响比一般条件转移指令还要大得多。这时,不仅要全部作废或部分作废先行指令缓冲栈中已经预取的指令,还可能要作废先行操作栈中的指令和先行读数栈中的操作数,作废当前在指令分析器中分析的指令。具体影响的程度要看先行控制器中所采用的策略,下一节中将具体分析可以采取的有关策略。

5.1.4.4 转移预测技术

从上面的分析中已经看到,条件转移指令对先行控制器的影响很大。而且,根据统计,在一般程序中,条件转移指令占五分之一左右。因此,在现代计算机系统中,对条件转移指令采用多种技术,以尽可能减少它对先行控制器产生的影响。在第二章的“RISC 关键技术”中已经介绍了延迟转移技术、指令取消技术等。在采用先行控制方式的处理机中,经常还采用如下几种技术:

1. 软件“猜测法”

从上两节的分析中已经看到,当转移不成功时,条件转移指令对先行控制器的影响比较小。因此,编译器在对源程序进行编译时,为了达到比较好的效果,要尽量降低转移成功出现的概率。

如图 5.21(a)所示的一个源程序,如果在一般编译器中进行编译,编译结果生成的目标程序,转移成功的概率很高,不成功的只有一次。这种编译结果对先行控制器的影响非常大。如果在编译器中进行适当的处理,编译成如图 5.21(b)所示的结果,转移成功与不成功的概率正好相反。这种编译结果很适合在一般的先行控制器中执行。例如,对于一个需要循环执行 1 000 次的程序,其中,999 次均转移不成功,对先行控制器影响很小,只有一次转移成功,对先行控制器有比较大的影响。如果处理机支持复合型条件转移指令,编译器也可以把源程序编译成如图 5.21(c)所示的结果。

软件“猜测法”的优点是不需要改变先行控制器的硬件结构,只要适当修改编译器就能够大幅度降低条件转移指令对先行控制器产生的影响,其最终目标是提高处理机执行程序的速度。

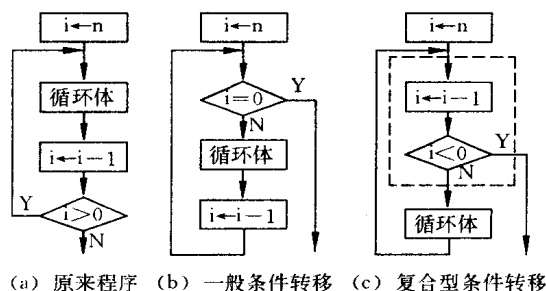


图 5.21 用编译器支持条件转移指令的执行

2. 硬件“猜测法”

如果修改编译器有困难,也可以通过改变先行控制器的硬件结构来降低转移指令对先行控制器造成的影响。这就是在很多机器中采用的硬件“猜测法”。

在先行指令缓冲栈的人口处再增设一个简单的指令分析器,当这个指令分析器检测到转移指令(包括无条件转移指令和条件转移指令)时,就把转移目标地址 L 送入先行程序计数器 PC_i 中,同时保留当前 PC_i 中的内容到另一寄存器中,以备猜错时恢复原来的程序执行方向。

当指令执行部件执行完形成条件码的指令之后,判断转移是否成功。如果转移成功,则猜测正确。这种情况下,条件转移或无条件转移指令几乎对先行控制器不造成影响。预取到先行指令缓冲栈中的指令,已经由指令分析器预处理完成,存放在先行操作栈中指令和已经读到先行读数栈中的操作数都有效。如果转移不成功,则猜测错误,这时,要用保存下来的地址恢复先行程序计数器 PC_i 和现行程序计数器 PC ,清除先行指令缓冲栈、先行操作栈和先行读数栈,重新开始取指令。

在先行控制器中使用“猜测法”,最好是软件和硬件共同配合,软件与硬件都往同一个方向去猜测。

3. 两个先行指令缓冲栈

上面介绍的两种“猜测法”都是针对循环程序的。由于控制循环的条件转移指令,其进入循环的概率要比出循环的概率大得多,因此,采用软件或硬件“猜测法”可以使转移指令

对先行控制器的影响减少到很小。然而,在实际的程序中,还有大量的普通条件转移指令,例如,由高级语言中的“IF”语句编译产生的条件转移指令。这种条件转移指令转移成功与不成功的概率一般各为 50%。对于这类条件转移指令,无论采用何种猜测法,其效果都是一样的。

一种比较有效的方法是:在先行指令缓冲栈中再增加一个先行目标缓冲栈。当指令分析器分析到条件转移指令时,按照转移成功方向预取指令到这个先行目标缓冲栈中。原来的先行指令缓冲栈仍然按照转移不成功方向继续预取指令。当指令分析器执行到条件转移指令时,等待指令执行部件送出转移条件码。如果转移不成功,则继续分析原来先行指令缓冲栈中指令。如果转移成功,则分析新增设的先行目标缓冲栈中的指令。图 5.22 就是采用先行指令缓冲栈时的条件转移指令执行流程。AIB 是新增加的先行目标缓冲栈,IB 是原来的指令缓冲栈。 TR_1 和 TR_2 是两个暂存寄存器,用来保存现行程序计数器 PC 和先行程序计数器 PC_1 中的内容,当转移不成功时,可以用保存在 TR_1 和 TR_2 中的内容恢复这两个程序计数器。

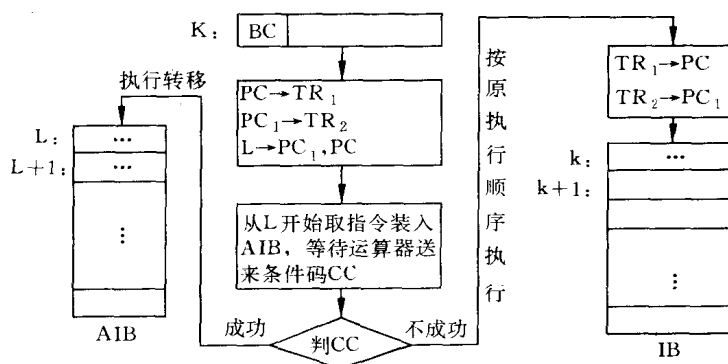


图 5.22 两个先行指令缓冲栈的条件转移指令执行流程

IBM 370/165 机就采用了两个先行指令缓冲栈。其中,先行目标缓冲栈 AIB 的缓冲深度为 4。另外,为了简化对两套先行指令缓冲栈的控制,在转移成功时,交换先行目标缓冲栈 AIB 和原来的先行指令缓冲栈 IB 中内容,使指令分析器仍然从原来的指令缓冲栈 IB 中取指令。

5.1.4.5 短循环程序的处理

在循环程序中存在有大量的短循环程序。对于短循环程序,一般先行指令缓冲栈的效率很低。一种极端的情况是:当循环体仅一条指令时,先行指令缓冲栈将根本不起作用。指令分析器分析完这条指令后,就清除指令缓冲栈,重新到主存储器中取这条指令。这样,反复取指令,又反复清除,实际上是一种很大的浪费。

那么,什么样的程序是短循环程序呢?在不同的机器中,短循环程序的定义有所不同。一般来说,短循环程序应该满足如下三个条件:

1. 循环体的长度小于等于先行指令缓冲栈的深度。使得在循环程序执行过程中,整个循环体能够始终存放在先行指令缓冲栈中不被清除,循环程序可以多次重复使用,以减

少访问主存的次数。

2. 循环次数的控制采用计数转移指令实现。因为计数转移指令可以在指令分析器中直接执行,因此,当指令分析器分析到这种特殊的条件转移指令时不必等待指令执行部件形成条件码,而在指令分析器中就可以判断循环出口条件是否满足。这样,可以完全消除控制循环的条件转移指令对先行控制器可能造成的不利影响。

3. 控制循环的条件转移指令一般是向后转移的指令。这是为了在判断是否是短循环程序时提供方便。

要在先行控制器中对短循环程序作特殊处理,必须解决好三个问题:一是指令分析器如何发现短循环程序,二是如何控制短循环程序在先行指令缓冲栈中不被清除,三是如何控制循环体的执行次数,即处理好循环程序的出口。下面介绍两种方法:

方法一:在指令系统中设置专门的短循环程序的开门指令和关门指令,由编译器在对源程序进行编译时发现短循环程序,并在短循环程序的开头加上短循环开门指令,在短循环程序的末尾加上短循环关门指令。在指令分析器中要增加专门的硬件来处理短循环开门指令和短循环关门指令,当指令分析器发现短循环开门指令时,就知道下面将要进入短循环程序。关门指令是整个循环程序的最后一条指令,用它来代替原来的计数转移指令控制循环的出口。为了控制存放在先行指令缓冲栈中短循环程序不被清除,要设置一个专门的短循环标志触发器 T_L 。当 $T_L=1$ 时,表示在先行指令缓冲栈中存放的是短循环程序,指令分析器在分析完短循环程序中的一条指令时,不清除先行指令缓冲栈中的指令。

在设置有专门的短循环开门指令和关门指令的处理机中,处理一个短循环程序的主要过程如下:

当指令分析器分析到短循环开门指令时,表示短循环程序将要开始。开门指令的主要功能是为短循环程序作预处理,主要做好三件事情:

1. 将短循环程序的循环次数送入指令分析器的循环次数计数器中。因为短循环程序的循环次数是由指令分析器控制的,即短循环程序开门指令和关门指令都在指令分析器内就执行完成。

2. 设置短循环程序在先行指令缓冲栈中的起始地址,即短循环程序的入口地址。有的机器规定短循环程序必须从先行指令缓冲栈的“0”地址开始存放;因此,开门指令要清除先行指令缓冲栈中的全部指令,并撤消已经发出的,但还没有得到主存响应的所有取指令请求,以保证后面从主存储器中读出的指令都是属于短循环程序的指令。但也有许多机器允许从先行指令缓冲栈的中间位置开始存放短循环程序,这时要保存短循环程序的入口地址,以便在执行完短循环程序的一个循环体时找到循环程序的入口。

3. 置位短循环标志触发器 T_L ,表示短循环程序开始。

在指令分析器执行完短循环开门指令以后,就从先行指令缓冲栈中取出一条指令来进行分析。这时,如果短循环标志触发器 $T_L=1$,每从先行指令缓冲栈中取出一条指令,先行指令缓冲栈中的指令都不被清除,从而使短循环程序能够保留在先行指令缓冲栈中以便下次还继续使用。

当指令分析器执行到关门指令时,表示短循环程序的一个循环体已经结束。这时,在先行指令缓冲栈中已经存放了短循环程序的全部指令。以后,每执行一次循环,指令分析

器都要执行一次关门指令。关门指令的主要功能是控制短循环程序的循环次数,当循环程序结束时能够正确地退出循环。它的主要工作是:

1. 把循环计数器的内容减“1”。

2. 若循环计数器中的新值大于或等于“1”,则把现行程序计数器指向短循环程序的入口,准备开始下一次循环。若循环计数器中的内容等于“1”,表示还要做最后一次循环,这时,要把短循环标志触发器 T_L 清除为“0”。当短循环标志触发器 $T_L=0$ 时,指令分析器每从先行指令缓冲栈中取出一条指令,先行指令缓冲栈中的这条指令就立即被清除。这样,先行指令缓冲栈恢复正常工作状态,又可以向存储控制器发出取指令的请求,为指令分析器分析短循环后面的程序作好准备。

3. 若循环计数器中的新值等于“0”,表示整个短循环程序已经执行完成,则顺序执行短循环程序以后的指令。

采用专门的短循环开门指令和关门指令的主要优点是硬件实现相对比较简单,因为,短循环程序的标识实际上是由程序员通过短循环开门指令和关门指令来做的。而且,短循环开门指令和关门指令是在指令分析器中执行的,通过指令的操作码就能识别这两条指令,不需要专门的识别硬件。当然,这种方法的主要缺点是增加了程序设计人员的负担,而且这种方法对程序员是不透明的。

在采用图 5.22 所示的两个先行指令缓冲栈的先行控制器中,可以把两个先行指令缓冲栈连接起来使用,以加大短循环程序的长度。

方法二:在有的机器中,为了使短循环程序的处理对程序员透明,不采用专门的短循环开门指令和关门指令,而是用专门的硬件来识别短循环程序。例如,在 IBM 360 和 IBM 370 系列机中就是这样做的。在 IBM 360/91 计算机中,指令分析器中有一个向后检测 8 条指令的功能。如果条件转移指令的转移目标地址与条件转移指令本身的地址之间相差小于“0”,且大于等于“-8”,则认为遇到了短循环程序。短循环程序的处理方法与设置专门的短循环开门指令和关门指令的方法相同。

对短循环程序进行专门处理后,可以加快短循环程序的执行速度,减少处理机与主存储器之间的通信量。据统计,短循环程序的执行时间可以缩短 $1/3$ 到 $3/4$ 。

目前,在许多高性能处理机中,采用专门的一级指令 Cache,即芯片内的一级 Cache,把指令 Cache 与数据 Cache 分开;而且 Cache 的存储容量比较大,通常为几个千字节至几十个千字节。由于在 Cache 中存放的指令可以长期保存,不像在先行指令缓冲栈中预取的指令,只执行一次之后就被清除,因此,采用大容量的高速一级指令 Cache,可以有效地提高循环程序的执行速度。

5.2 流水线处理机

可以从两个方面来开发处理机内部的并行性,一个是所谓空间并行性,即在一个处理机内设置多个独立的操作部件,并让这些操作部件并行工作,这种处理机称为多操作部件处理机和超标量处理机,这部分内容将在下一节中介绍;另一个是所谓时间并行性,就是采用流水线技术。流水线技术是一种非常经济、对提高处理机的运算速度非常有效的技

术。采用流水线技术可以不增加硬件或只需要增加少量硬件就能够把处理机的运算速度提高几倍,它是目前使用非常普遍的一种并行处理方式。本节首先介绍流水线的基本原理、特点、分类、性能分析等,然后是非线性流水线的调度问题,最后介绍流水线中的局部相关和全局相关的处理方法。

5.2.1 流水线工作原理

流水线方式是把一个重复的过程分解为若干个子过程,每个子过程可以与其他子过程同时进行。由于这种工作方式与工厂中的生产流水线十分相似,因此,把它称为流水线工作方式。

在处理机的各个部分几乎都可以采用流水线方式工作。指令的执行过程可以采用流水线,称为指令流水线。运算器中的操作部件,如浮点加法器、浮点乘法器等可以采用流水线,称为操作部件流水线。访问主存储器部件也可以采用流水线。甚至在处理机之间,机器之间也可以采用流水线。

5.2.1.1 从重叠到流水线

上一节中介绍的一次重叠执行方式就是一种简单的指令流水线。在采用先行指令缓冲栈的处理机中,一条指令的执行过程可以比较粗地分解为“分析”和“执行”两个子过程,这两个子过程分别在指令分析器和指令执行部件中完成,如图 5.23 所示。由于在指令分析器和指令执行部件的输出端各有一个锁存器,可以分别保存指令“分析”和指令“执行”的结果,因此,指令分析器和指令执行部件能够成为两个完全独立的功能部件,它们可以同时并行工作,不必等待的“分析 k ”和“执行 k ”两个子过程都完成之后,指令分析器才开始“分析 $k+1$ ”这个新的子过程。实际上,在指令分析器“分析 k ”刚刚结束,并将分析结果送入指令执行部件开始“执行 k ”这一子过程的同时,指令分析器就可以开始“分析 $k+1$ ”这个新的子过程。因此,指令分析器“分析 $k+1$ ”与指令执行部件“执行 k ”可以同时进行。

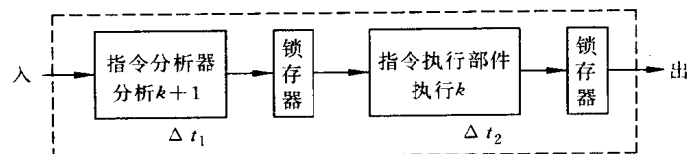


图 5.23 一种简单的流水线

在图 5.23 中,如果指令分析器分析一条指令所需要的时间 Δt_1 与指令执行部件执行一条指令所需要的时间 Δt_2 相等,即 $\Delta t_1 = \Delta t_2$,都为 Δt 。尽管处理机执行一条指令仍然需要 $2\Delta t$ 的时间才能完成,然而,由于指令分析部件与指令执行部件是并行工作的,因此,从指令执行部件的输出端看,每间隔一个 Δt 就执行完成一条指令,并输出一个运算结果。因此,处理机执行指令的速度提高了一倍。

如果把执行一条指令的过程分得更细,如图 5.24 所示,可以分为 6 个子过程。当然,每一个部件的输出端都要有一个锁存器。

在图 5.24 中的每一个子过程还可以再进一步分解成更小的子过程,即在功能部件的

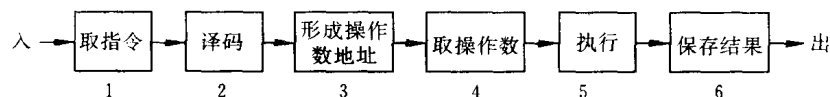


图 5.24 一种指令流水线

内部也采用流水线方式工作。例如，一个浮点加法的执行过程可以采用 3 级至 6 级，甚至更多级的流水线。图 5.25 是一个浮点加法器的 4 级流水线，它将浮点加法的全过程分解为求阶差、对阶、尾数加和规格化 4 个小子过程，每一个小子过程可以在各自独立的功能部件上完成。

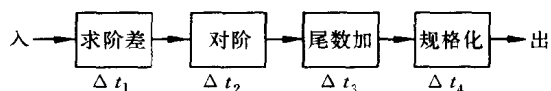


图 5.25 浮点加法器流水线

图 5.25 中，每一个部件的输出端都有一个锁存器，用于保存本部件的执行结果。如果各个部件的执行时间均相等，即 $\Delta t_1 = \Delta t_2 = \Delta t_3 = \Delta t_4 = \Delta t$ ，虽然执行一次浮点加法的时间仍然需要 $4\Delta t$ ，然而，由于 4 个部件同时工作，每隔一个 Δt 就能够完成一次浮点加法，输出一个运算结果。因此，采用 4 级流水线的浮点加法器，处理机执行浮点加法的速度能够提高 3 倍。

5.2.1.2 时空图

描述流水线的工作，最常用的方法是采用“时空图”。例如，图 5.23 所示的一条简单流水线，采用时空图表示如图 5.26 所示。

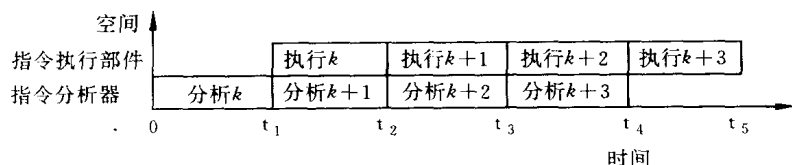


图 5.26 描述流水线工作过程的时空图

在时空图中，横坐标表示时间，也就是输入到流水线中的各个任务在流水线中所经过的时间。当流水线中的各个功能部件的执行时间都相等时，横坐标被分割成相等长度的时间段。纵坐标表示空间，即流水线的各个子过程。在时空图中，流水线的的一个子过程通常称为“功能段”。

图 5.27 是一个 4 段浮点加法器流水线的时空图。由求阶差、对阶、尾数加和规格化 4 个功能部件同时工作来完成浮点加法。

从图 5.27 的流水线时空图中，能够很清楚地看出各个任务在流水线的各段中流动的过程。从横坐标方向看，流水线中的各个功能部件在逐个连续地完成自己的任务，例如，求阶差部件在完成“求阶差 1”任务之后，紧接着做“求阶差 2”、“求阶差 3”……，同样，对阶

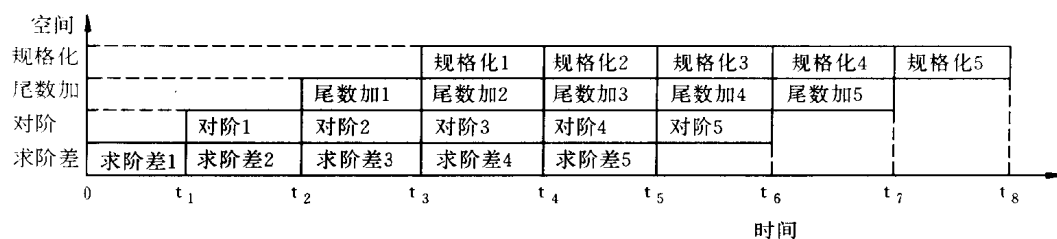


图 5.27 浮点加法器流水线的时空图

部件在完成“对阶 1”任务之后,接着做“对阶 2”、“对阶 3”……。从纵坐标方向看,在同一个时间段内有多的功能段在同时工作;例如,在 $t_3 \sim t_4$ 这一个时间段内,规格化部件在做第 1 个加法的规格化,尾数加部件在做第 2 个加法的尾数加,对阶部件在做第 3 个加法的对阶,求阶差部件在做第 4 个加法的求阶差。加法器的 4 个独立的功能部件同时工作,分别做 4 个加法的不同阶段。因此,时空图是描述流水线工作过程的一种比较好的方法。

5.2.1.3 流水线的特点

从上面的分析中可以看到,在处理器中采用流水线方式与采用传统的串行方式相比,具有如下特点:

1. 在流水线中处理的必须是连续任务,只有连续不断地提供任务才能充分发挥流水线的效率。例如,要使浮点加法器流水线充分发挥作用,需要连续提供浮点加法运算。然而,由于程序本身的原因和程序设计过程中人为造成的原因,如数据相关等,不可能连续为浮点加法器提供同一种操作。因此,在采用流水线方式工作的处理器中,特别是当流水线的级数较多时,要在软件和硬件设计等多方面尽量为流水线提供连续的任务,以提高流水线的效率。

2. 把一个任务(一条指令或一个操作)分解为几个有联系的子任务,每个子任务由一个专门的功能部件来实现。因此,流水线实际上是把一个大的功能部件分解为多个独立的功能部件,并依靠多个功能部件并行工作来缩短程序的执行时间。在流水线中,一个子任务通常称为一个子过程,或流水线中的一个功能段。

3. 在流水线的每一个功能部件的后面都要有一个缓冲寄存器,或称为锁存器、闸门寄存器等,用于保存本段的执行结果。这是因为流水线中每一段的延迟时间一般不可能都相等,因此,在段与段之间传送子任务时,必须通过缓冲寄存器。当某一个功能段的执行时间变化范围比较大时,要设置多个缓冲寄存器。

4. 流水线中各段的时间应尽量相等,否则将引起“堵塞”、“断流”等。执行时间长的一段将成为整个流水线的“瓶颈”,这时,流水线中的各个功能部件将不能充分发挥作用。因此,在流水线设计中,当遇到“瓶颈”时,必须采取办法解决。

5. 流水线需要有“装入时间”和“排空时间”。只有流水线完全充满时,整个流水线的效率才能得到充分发挥。

在流水线处理机的设计过程中,要充分注意上述问题,以设计出高效率的流水线。在流水线处理机上设计程序时,也必须注意流水线的上述特点,以充分发挥流水线处理机的

高效率。

5.2.2 流水线的分类

从不同的角度,按照不同的观点,可以把流水线分成多种不同的种类。平时所说的某种流水线,往往是按照某种观点,或从某一个特定角度对流水线进行分类的结果,因此,从名称上只能反映这种流水线在某一方面的特点或性能。

5.2.2.1 线性流水线与非线性流水线

按照流水线的各个功能段之间是否有反馈信号,可以把流水线分为线性流水线和非线性流水线两类。

线性流水线(linear pipelining)是将流水线的各段逐个串接起来。输入数据从流水线的一端进入,从另一端输出。数据在流水线中的各个功能段流过时,每一个功能段都流过一次,而且仅仅流过一次。

一条线性流水线通常只完成一种固定的功能。在现代计算机系统中,线性流水线已经被非常广泛地应用于指令执行过程、各种算术运算操作、存储器访问操作等。在上一节中介绍的流水线中,如图 5.23、图 5.24 所示的指令流水线,图 5.25 所示的浮点加法器流水线等都属于线性流水线。

图 5.28 是一种简单的非线性流水线(nonlinear pipelining)。在流水线的各个功能段之间除了有串行的连接之外,还有反馈回路。在图 5.28 中,流水线的功能段 S₂ 的输出可能直接传送给功能段 S₃,也可能反馈到本功能段的输入。

在图 5.28 中,虽然总共只有三个功能段,但是,输入任务经过流水线到达输出,往往不只是经过三个时钟周期;其中,功能段 S₂ 可能要被多次调用,这也是非线性流水线与线性流水线的区别。因此,在非线性流水线中,只用图 5.28 这样一种连接图并不能表示出一个任务在流水线中实际流动的过程。

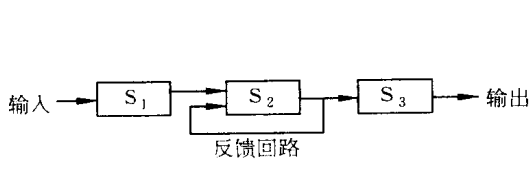


图 5.28 一种简单的非线性流水线

	时间			
	1	2	3	4
S ₁	×			
S ₂		×	×	
S ₃				×

图 5.29 非线性流水线的预约表

非线性流水线经常用于递归调用,或构成多功能流水线等。表示非线性流水线的工作情况除了要图 5.28 这种流水线的连接图之外,通常还需要一张“预约表”,用两者共同来表示流水线的工作情况。在预约表中可以很清楚地表示出反馈回路的使用次数。例如,图 5.29 表示图 5.28 的中反馈回路仅使用一次的预约表,图中用“×”表示这一个功能段在相应的这一段时间内有效,即任务经过了这一个功能段。

一条非线性流水线可以对应有很多张预约表,同样,一张预约表实际上仅表示了一条非线性流水线的一种工作方式。关于非线性流水线中的一些更加深入的问题,将在本章的

非线性流水线调度一节中作比较详细的介绍。

线性流水线实际上也有预约表,只不过它的预约表是确定的。首先,预约表的水平方向与垂直方向的格数一定是相等,即组成一个正方形。其次,预约表中从左上角到右下角所有格子是全部有效的,即是打“×”的;而预约表的其余部分一定都是空白的。因此,在描述线性流水线时,一般不给出预约表。

非线性流水线调度中的一个重要问题是要确定在什么时间可以向流水线输入新的任务,使新输入任务与流水线中原来的反馈任务之间在各个功能段上都不产生冲突。这就是非线性流水线的调度问题,这个问题将在下面的专门一节中介绍。

5.2.2.2 流水线的级别

按照流水线使用的不同级别,可以把流水线分为功能部件级、处理机级和处理机间级等多种类型。

处理机级流水线又称为指令流水线(instruction pipelining)。它把一条指令的执行过程分解为多个子过程,每个子过程在一个独立的功能部件中完成。前面介绍的一次重叠执行方式是在设置有先行指令缓冲栈的处理机中采用的一种简单的指令流水线。它把一条指令的执行过程分解为“分析”和“执行”两个子过程。因此,一次重叠执行方式可以同时执行两条指令。

在采用先行控制器的处理机中,组成先行控制器的各个部件实际上也构成了一条流水线。如图 5.30 所示,在先行控制器中,一条指令的执行过程被分解为 5 个子过程,每个子过程在一个专用的功能部件中执行。由于各种指令在同一个功能部件中执行的时间往往相差很大,因此,在每一个功能段之间要设置多个缓冲寄存器,以平滑流水线中各个功能部件的操作。

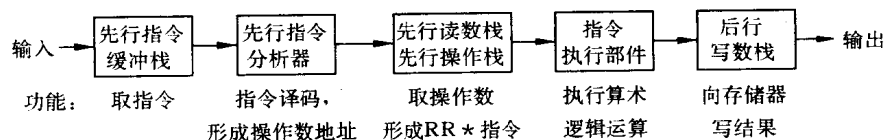


图 5.30 先行控制方式中的指令流水线

在图 5.30 中,每一个部件内部还可以采用流水线来实现。例如,对于一些比较复杂的运算操作部件,如浮点加法器、浮点乘法器等,一般要采用多级流水线来实现。后行写数栈和先行读数栈也可以采用多级流水线来实现。这种流水线被称为部件级流水线,或功能部件级流水线。

功能部件级流水线也称为运算操作流水线(arithmetic pipelining)。图 5.25 中的浮点加法器流水线就是一种典型的功能部件级流水线。后面,还将介绍几种常见的功能部件级流水线。

要提高执行部件执行算术逻辑运算操作的速度,除了在运算操作部件中采用流水线之外,也可以设置多个独立的操作部件,并通过这些操作部件的并行工作来提高处理机执行算术逻辑运算的速度。通常,把指令执行部件中采用了流水线的处理机称为流水线处理

机或超流水线处理机,而把指令执行部件中设置有多多个操作部件的处理机称为多操作部件处理机或超标量处理机。

处理机间流水线又称为宏流水线(macro pipelining),如图 5.31 所示。这种流水线由两个或两个以上处理机通过存储器串行连接起来,每个处理机对同一个数据流的不同部分分别进行处理。前一个处理机的输出结果存入存储器中,作为后一个处理机的输入,每个处理机完成整个任务中的一部分。

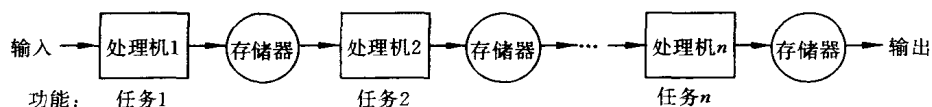


图 5.31 一种宏流水线

一台大型计算机系统通常由多个同型号的或不同型号的处理机构成,每个处理机有不同的分工。例如,有多个用高级语言编写的程序需要在机器上运行,则程序和数据的输入、编译、连接、执行、执行结果输出等可以分别在不同的处理机上完成,这些处理机就构成了一条宏流水线。

5.2.2.3 单功能与多功能流水线

一条流水线只能完成一种固定的功能,这种流水线称为单功能流水线(unifunction pipelining)。例如,浮点加法器流水线专门完成浮点加法运算,浮点乘法器流水线专门完成浮点乘法运算。当要实现多种不同功能时,可以采用多条单功能流水线。如 Cray-1 计算机中有 12 条单功能流水线,我国研制的 YH-1 计算机有 18 条单功能流水线。Pentium 处理机有一条 5 段的整数运算流水线和一条 8 段的浮点运算流水线。采用超流水线体系结构的 Alpha 21064 处理机有三条流水线,其中,整数操作和访问存储器操作为 7 段流水线,浮点运算操作为 10 段流水线。

多功能流水线(multifunction pipelining)是指流水线的各段可以进行不同的连接。在不同时间内,或在同一时间内,通过不同的连接方式实现不同的功能。

多功能流水线的典型代表是 Texas 仪器公司的高级科学计算机 ASC 中采用的 8 段流水线。在一台 ASC 处理机内有 4 条相同的流水线,每条流水线通过不同的连接方式可以完成整数加减法运算、整数乘法运算、浮点加法运算、浮点乘法运算,还可以实现逻辑运算、移位操作和数据转换功能等。它除了支持标量运算之外,还支持向量运算,如两个向量的浮点点积运算等。

图 5.32(a)是流水线的连接关系,实现不同的运算要求使用流水线中的不同功能部件,并在功能部件之间建立不同的连接关系。因此,要求多功能流水线能够根据运算需要在有的部件之间建立连接,而在另一些部件之间不建立连接。图 5.32(b)是实现定点乘法时的流水线连接关系,它只用了流水线中的 4 个功能部件,有阴影的另外 4 个功能部件不用。图 5.32(c)是实现浮点加法或浮点减法时的流水线连接关系,它用了 6 个功能部件,也就是说,实现浮点加法或减法采用的是 6 级流水线。图 5.32(d)是对两个浮点向量求点积的流水线连接关系,它实际上是一条带有反馈回路的非线性流水线。两个浮点向量

A 与 B 的点积计算公式为:

$$f = \sum_{i=1}^n A_i \times B_i \quad (5.20)$$

求点积的主要运算是“乘-加”操作。要完成关系式(5.20)的计算要做 n 次“乘-加”运算,要反复多次使用流水线。

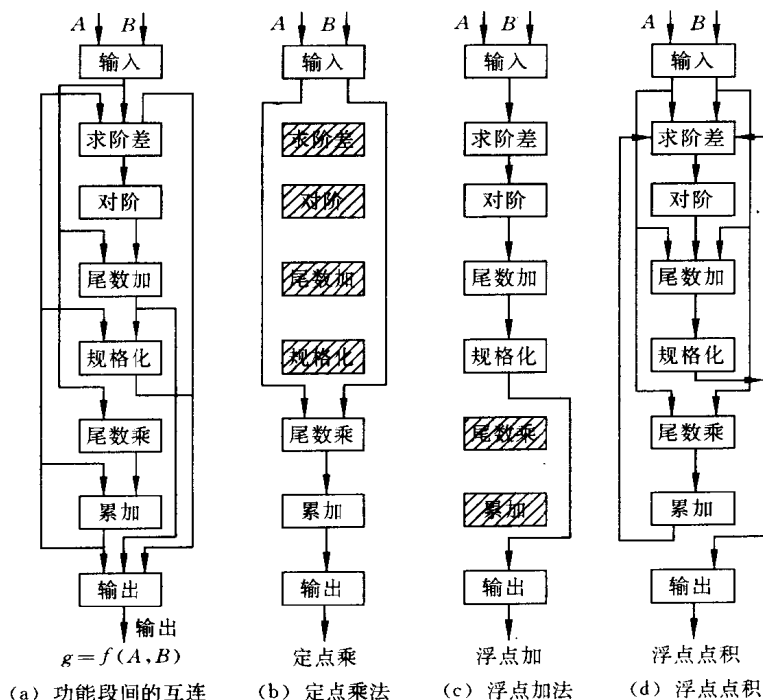


图 5.32 TI-ASC 计算机的多功能流水线

在处理机中采用多功能流水线的优点是流水线中各个功能部件的利用率比较高。由于在实际的标量运算程序中,各种运算操作一般是混合在一起的,这一点与向量运算操作有很大的不同,因此,在标量计算机的指令执行部件中采用多功能流水线是一种比较合理的选择。与采用多功能流水线不同的另一种方案是设置多条专门的单功能流水线,在许多向量流水线处理机中就是这样做的,例如,在我国研制的 YH-1 向量计算机就设置有 18 条单功能流水线。

5.2.2.4 静态流水线与动态流水线

在多功能流水线中,按照在同一时间内是否能够连接成多种方式,同时执行多种功能,可以把多功能流水线分为静态流水线和动态流水线两种。所谓静态流水线(static pipelining)是指在同一段时间内,多功能流水线中的各个功能段只能按照一种固定的方式连接,实现一种固定的功能。只有当按照这种连接方式工作的所有任务都流出流水线之后,多功能流水线才能重新进行连接,以实现其它功能。例如,图 5.32 中的 8 段多功能流水线,如果按照图 5.33 所示的时空图工作,就是一种静态流水线。开始时,多功能流水线

按照实现浮点加减法的方式连接,当 n 个浮点加减法全部执行完成,最后一个浮点加减法运算的排空操作也做完之后,多功能流水线才重新开始按照实现定点乘法的方式连接,并开始做定点乘法运算。

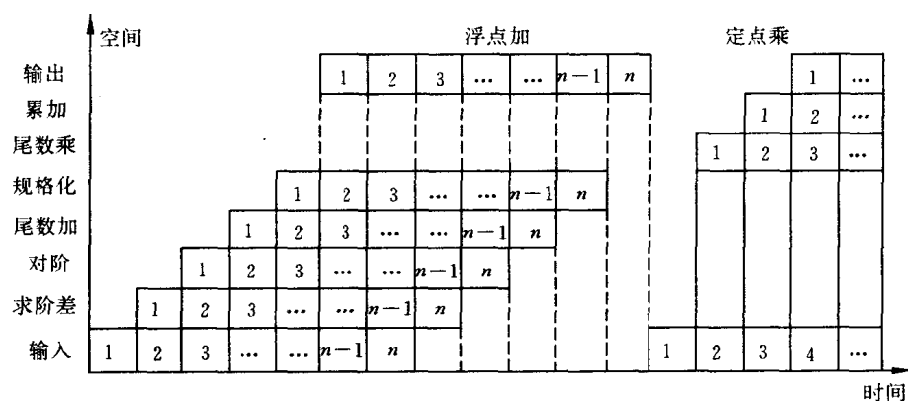


图 5.33 静态流水线时空图

动态流水线(dynamic pipelining)是指在同一段时间内,多功能流水线中的各段可以按照不同的方式连接,同时执行多种功能。当然,同时实现多种连接方式是有条件的,即流水线中的各个功能部件之间不能发生冲突。例如,在图 5.34 中,中间有一段时间,在同一条多功能流水线的不同功能段中在同时执行浮点加减法和定点乘法两种运算。也就是说,当浮点加减法运算还没有全部完成时,定点乘法运算就已经开始了。两种运算同时在同一条多功能流水线中分别使用不同的功能段。

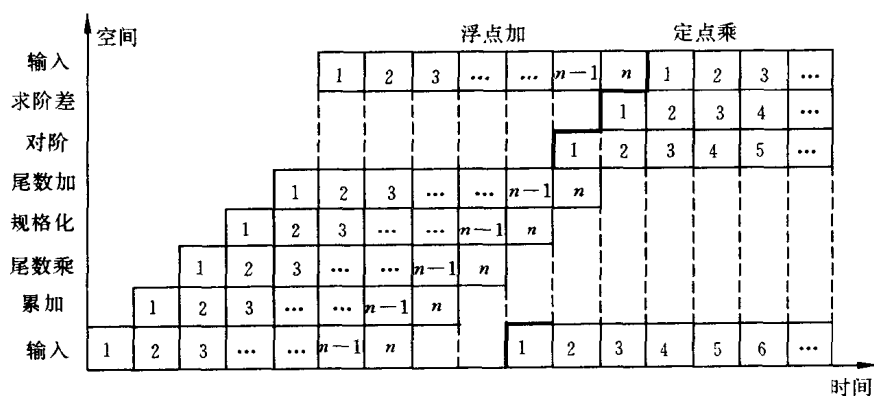


图 5.34 动态流水线时空图

在静态流水线中,只有程序中连续出现同一种运算时,流水线的效率才能得到充分的发挥。如果输入到流水线中的是一串不同运算相互间隔的操作,例如,输入的一串操作是浮点加、定点乘、浮点加、定点乘……,则这条静态流水线的效率就与顺序执行方式完全一样。而动态流水线则不同,它允许两种运算在同一条流水线中同时执行。因此,在一般情况下,动态流水线的效率和功能部件的利用率要比静态流水线高,但是,动态流水线的控

制比静态流水线要复杂得多。目前,在大多数处理机中均采用静态流水线。

除了以上几种流水线的分类方法之外,还可以从其它多种不同角度来划分流水线。例如,按照不同的数据表示方式,可以把流水线分为标量流水线和向量流水线两种。标量流水线一般用于标量处理机中,而向量流水线主要用于向量计算机中。在本章中介绍的都是标量流水线。

在线性流水线中,根据对流水线的控制方式不同,可以把流水线分为同步流水线和异步流水线两类。在本章中介绍的都是同步流水线。一般的宏流水线多采用异步流水线方式。如图 5.35 所示,在异步流水线中,当 S_i 功能段要向 S_{i+1} 功能段传送数据时,首先发出就绪信号, S_{i+1} 功能段收到就绪信号后,向 S_i 回送一个回答信号。

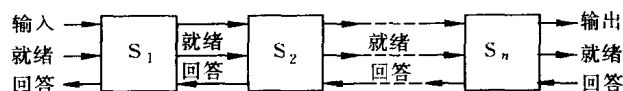


图 5.35 一种异步流水线

按照流水线输出端流出的任务与流水线输入端流入的任务的顺序是否相同,可以把流水线分为顺序流水线与乱序流水线两种。乱序流水线在有的资料上又称为无序流水线、错序流水线或异步流水线等。

在顺序流水线中,流水线输出端的任务流出顺序与输入端的任务流入顺序完全相同,每个任务在流水线的各个功能段中是一个跟着一个顺序流动的。而乱序流水线输出端流出任务的顺序与输入端流入任务的顺序可以不一样。每个任务在流水线中并不是按照输入的顺序一个跟着一个流动的。例如,在指令流水线中,有的指令要到主存储器中读操作数,也有的指令不需要到主存储器中读操作数。由于到主存储器中读操作数可能需要等待比较长的一段时间,因此,在不发生数据相关的情况下,只要指令执行部件有空闲,不需要到主存储器中读操作数的指令可以超越需要等待从主存储器中读出操作数的指令先在指令执行部件中执行。

5.2.3 线性流水线的性能分析

以下,以线性流水线为例,分析流水线的主要性能。其分析方法和有关公式也适用于非线性流水线。

衡量流水线性能的主要指标有吞吐率、加速比和效率。另外,在流水线设计中,如何选择流水线的最佳段数也是一个非常重要的问题。因此,本节首先对影响流水线性能的吞吐率、加速比和效率等三个主要指标进行分析,然后介绍流水线最佳段数的选择方法,最后举例说明流水线的性能分析方法。

5.2.3.1 吞吐率

流水线的吞吐率(throughput rate, TP)是指在单位时间内流水线所完成的任务数量或输出的结果数量。

$$TP = \frac{n}{T_k} \quad (5.21)$$

式中, n 为任务数, T_k 是处理完成 n 个任务所用的时间。(5.21)式是计算流水线吞吐率的最基本公式。以下讨论满足某种特殊情况的流水线吞吐率。

如图 5.36 所示,在流水线各段的执行时间均相等,输入到流水线中的任务是连续的理想情况下,一条 k 段线性流水线能够在 $k + n - 1$ 个时钟周期内完成 n 个任务。可以从两个方面来分析流水线完成 n 个任务所需要的总时间。一种分析方法是从流水线的输出端看,用 k 个时钟周期输出第一个任务,其余 $n - 1$ 个时钟周期,每个周期输出一个任务,即用 $n - 1$ 个时钟周期输出 $n - 1$ 个任务。另一种分析方法是从流水线的输入端看,用 n 个时钟周期向流水线输入 n 个任务,另外还要用 $k - 1$ 个时钟周期作为流水线的排空时间。因此,流水线完成 n 个连续任务需要的总时间为:

$$T_k = (k + n - 1)\Delta t \quad (5.22)$$

其中, Δt 为时钟周期。

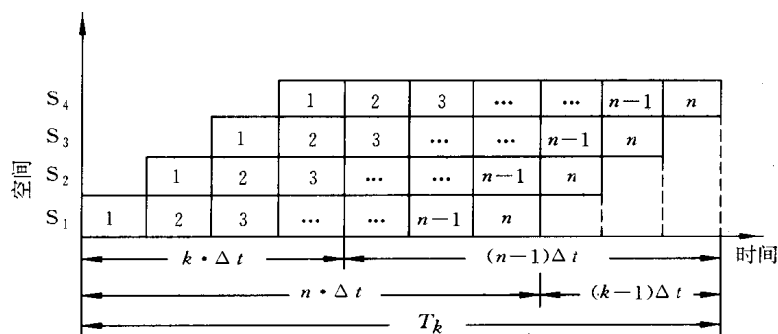


图 5.36 各段执行时间均相等的流水线时空图

把(5.22)关系式代入(5.21)式中,得到流水线各段执行时间均相等,输入连续 n 个任务的一条 k 段线性流水线的实际吞吐率为:

$$TP = \frac{n}{(k + n - 1)\Delta t} \quad (5.23)$$

这种情况下的最大吞吐率为:

$$TP_{\max} = \lim_{n \rightarrow \infty} \frac{n}{(k + n - 1)\Delta t} = \frac{1}{\Delta t} \quad (5.24)$$

最大吞吐率与实际吞吐率的关系是:

$$TP = \frac{n}{k + n - 1} TP_{\max} \quad (5.25)$$

从(5.25)式中可以看出来,流水线的实际吞吐率要小于最大吞吐率,它除了与时钟周期 Δt 有关之外,还与流水线的段数 k 、输入到流水线中的任务数 n 等有关。只有当 $n \gg k$ 时,才有 $TP \approx TP_{\max}$ 。

当流水线中各段的执行时间不完全相等时,流水线中就存在有“瓶颈”。如图 5.37(a) 所示,一个 4 段流水线中,第 2 段的执行时间是其他各段执行时间的 3 倍,即 $\Delta t_2 = 3\Delta t_1 = 3\Delta t_3 = 3\Delta t_4 = 3\Delta t$ 。在这种情况下的流水线时空图如图 5.37(b) 所示,图中的阴影部分表示该段流水线在这一段时间内是空闲不用的。因此,流水线各段执行时间不相等情况下的

实际吞吐率为：

$$TP = \frac{n}{\sum_{i=1}^k \Delta t_i + (n-1)\max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)} \quad (5.26)$$

分母中的第一部分是流水线完成第一个任务所用时间，第二部分是完成其余 $n-1$ 个任务所用的时间。

这时候流水线的最大吞吐率为：

$$TP_{\max} = \frac{1}{\max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)} \quad (5.27)$$

对于图 5.37 所示的例子，流水线的最大吞吐率为：

$$TP_{\max} = \frac{1}{3 \cdot \Delta t} \quad (5.28)$$

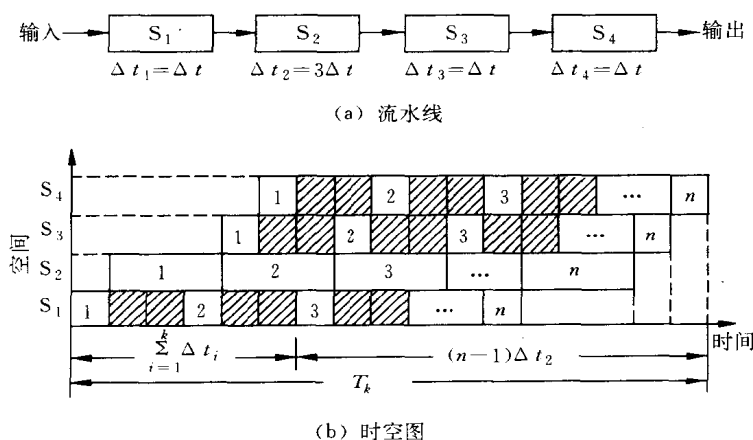


图 5.37 各段执行时间不相等的流水线及其时空图

从关系式(5.26)和(5.27)中看到，当流水线中各个功能段的执行时间不完全相等时，流水线的最大吞吐率与实际吞吐率主要是由流水线中执行时间最长的那个功能段来决定的，这个功能段就成了整个流水线的“瓶颈”。从图 5.37(b)中也可以看到，除了流水线中的“瓶颈”段一直处于忙碌状态外，其余各段有许多时间是空闲的，这实际上是一种资源的浪费。

解决流水线“瓶颈”问题的方法主要有两种。一种方法是将流水线的“瓶颈”部分再细分。如图 5.38 所示，把第二个功能段再细分为 3 个子功能段，分别为 S_{2-1} 、 S_{2-2} 、 S_{2-3} 。这样，每一个功能段及子功能段的延迟时间均为 Δt 。

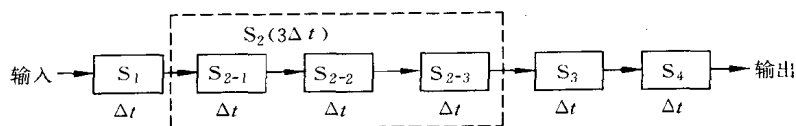


图 5.38 “瓶颈”功能段再次细分

如果由于结构等方面的原因,瓶颈功能段不能再细分时,可以采用如图 5.39 的方法,通过重复设置瓶颈功能段,让多个瓶颈功能段并行工作。对于图 5.38 和图 5.39 两种情况,流水线的最大吞吐率仍然可以达到:

$$TP_{\max} = \frac{1}{\Delta t} \quad (5.29)$$

但是,采用瓶颈功能段重复设置的方法,其控制逻辑比较复杂。例如,图 5.39(a)中,从功能段 S_1 到功能段 S_2 的各个并列功能段之间要设置一个数据分配器。从多个并列功能段 S_2 到功能段 S_3 之间要设置数据收集器。数据分配器的任务是:从功能段 S_1 输出的第 1 个任务分配给功能段 S_{2-1} ,输出的第 2 个任务分配给功能段 S_{2-2} ,输出的第 3 个任务分配给功能段 S_{2-3} ,以后依次重复。同样,数据收集器的任务是依次从三个功能段 S_{2-1} 、 S_{2-2} 和 S_{2-3} 收集处理结果,并分时输入到功能段 S_3 中。瓶颈功能段重复设置的流水线时空图如图 5.39(b)所示。

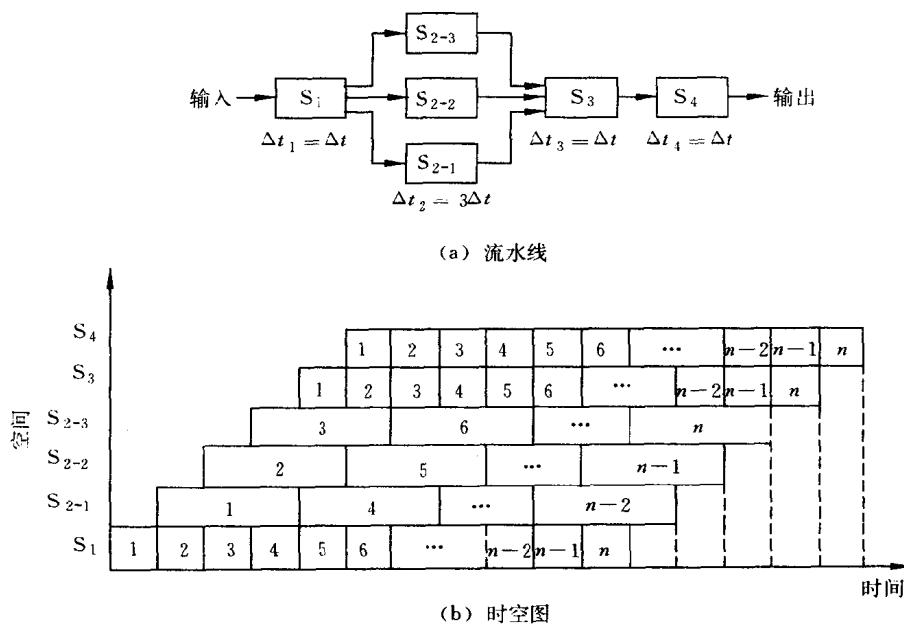


图 5.39 “瓶颈”功能段重复设置的流水线

实际上,由于存在有多种原因,使流水线的实际吞吐率要低于最大吞吐率。例如,流水线存在有装入和排空时间,输入的任务往往不是连续的,程序本身存在有数据相关,多功能流水线在完成某一种功能时有的功能段不使用等。必须注意,计算流水线实际吞吐率的基本公式是(5.21)式,其余关系式都是在特殊情况下推导出来的,在使用有关公式时,要特别注意这些公式的使用条件。

5.2.3.2 加速比

完成一批任务,不使用流水线所用的时间与使用流水线所用的时间之比称为流水线的加速比(speedup ratio)。如果不使用流水线,即顺序执行所用的时间为 T_0 ,使用流水线

的执行时间为 T_k , 则流水线的加速比为:

$$S = \frac{T_0}{T_k} \quad (5.30)$$

这是计算流水线加速比的基本公式。

如果流水线各个功能段的执行时间都相等, 则一条 k 段流水线完成 n 个连续任务所需要的时间如(5.22)式所示。如果不使用流水线, 即顺序执行这 n 个任务, 则所需要的时间为: $n k \Delta t$ 。因此, 各个功能段执行时间均相等的一条 k 段流水线完成 n 个连续任务时的实际加速比为:

$$S = \frac{k \cdot n \cdot \Delta t}{(k + n - 1) \Delta t} = \frac{k \cdot n}{k + n - 1} \quad (5.31)$$

这种情况下的最大加速比为:

$$S_{\max} = \lim_{n \rightarrow \infty} \frac{k \cdot n}{k + n - 1} = k \quad (5.32)$$

从(5.32)式中可以看出, 当 $n \gg k$ 时, 在线性流水线的各段执行时间均相等的情况下, 流水线的最大加速比等于流水线的段数。

那么, 是否流水线的段数愈多愈好呢? 实际上, 当流水线的段数很多时, 为了使流水线能够充分发挥效率, 要求连续输入的任务数 n 也就很多。图 5.40 给出连续任务个数 n 与加速比 S 的关系。当任务个数很小时, 加速比可能很差, 当 $n = 1$ 时, 加速比 S 的值最小为 1。当流水线的段数 k 增大时, 可以获得比较好的加速比。当 $n = 64$ 时, 一条 4 段流水线的加速比为 3.8, 而一条 8 段流水线的加速比可以达到 7.2。然而, 一方面, 由于一般程序中存在有数据相关、转移、中断等情况, 连续输入的任务数 n 受到很大的限制。另一方面, 由于控制的复杂性、电路实现及组装技术、实现的成本等方面的限制, 流水线的段数也不可能很多。

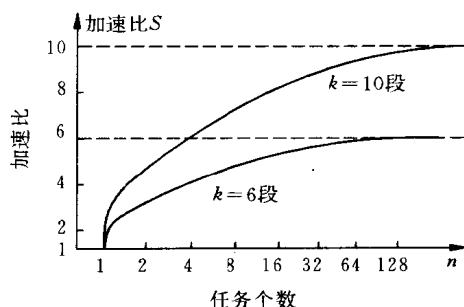


图 5.40 任务个数与加速比的关系

当流水线的各个功能段的执行时间不相等时, 一条 k 段线性流水线完成 n 个连续任务的实际加速比为:

$$S = \frac{n \cdot \sum_{i=1}^k \Delta t_i}{\sum_{i=1}^k \Delta t_i + (n - 1) \cdot \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)} \quad (5.33)$$

5.2.3.3 效率

流水线的效率(efficiency)是指流水线的设备利用率。在时空图上, 流水线的效率定义为 n 个任务占用的时空区与 k 个功能段总的时空区之比。因此, 流水线的效率包含有时间和空间两方面的因素。实际上, n 个任务占用的时空区就是顺序执行 n 个任务所使用的

总的时间 T_0 。而用一条 k 段流水线完成 n 个任务的总的时空区为 $k T_k$, 其中, T_k 是流水线完成 n 个任务所使用的总时间。则一条 k 段流水线的效率可以表示为:

$$E = \frac{n \text{ 个任务占用的时空区}}{k \text{ 个流水段的总的时空区}} = \frac{T_0}{k \cdot T_k} \quad (5.34)$$

(5.34)式是计算流水线效率的一般公式。

如果流水线的各段执行时间均相等,而且输入的 n 个任务是连续的,则一条 k 段流水线的效率为:

$$E = \frac{k \cdot n \cdot \Delta t}{k \cdot (k + n - 1) \cdot \Delta t} = \frac{n}{k + n - 1} \quad (5.35)$$

从流水线的时空图中看,(5.35)式的分母部分是完成 n 个任务所用的时间与 k 个功能段所围成的总面积,而分子部分是 n 个任务实际上占用的有效面积。因此,通过时空图来计算流水线的效率非常方便。

在流水线的各段执行时间均相等,输入给流水线的任务是连续的情况下,流水线的最高效率为:

$$E_{\max} = \lim_{n \rightarrow \infty} \frac{n}{k + n - 1} = 1 \quad (5.36)$$

从这个关系式中可以看出,当 $n \gg k$ 时,流水线的效率达到最大值 1。这时,流水线的各段均处于忙碌状态。从时空图中看,每一块都是有效的。

从(5.35)和(5.23)两个关系式中,很容易得出:

$$E = TP \cdot \Delta t \quad \text{或} \quad TP = \frac{E}{\Delta t} \quad (5.37)$$

当时钟周期 Δt 不变时,流水线的效率与吞吐率成正比。这就是说,为了提高流水线的效率而采取的措施,同时也提高了流水线的吞吐率。

比较(5.35)和(5.31)两个关系式,可以得出:

$$E = \frac{S}{k} \quad \text{或} \quad S = k \cdot E \quad (5.38)$$

流水线的效率是流水线实际加速比 S 与它的最大加速比 k 之比。只有当流水线的效率达到其最大值,即 $E = 1$ 时,才能使实际加速比达到最大,即 $S = k$ 。

如果流水线的各段执行时间不相等,参照图 5.37 和关系式(5.33),可以得出连续执行 n 个任务时的流水线效率为:

$$E = \frac{n \cdot \sum_{i=1}^k \Delta t_i}{k \cdot \left[\sum_{i=1}^k \Delta t_i + (n-1) \cdot \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k) \right]} \quad (5.39)$$

在这种情况下,流水线中除了瓶颈功能段之外,其它各个功能段都有空闲时间,这些功能段的效率没有得到充分发挥,因此,整个流水线的效率 E 也比较低。在图 5.38 和图 5.39 中采用的两种提高流水线吞吐率的方法,由于能够使流水线中的各个功能段始终处于忙碌状态,没有空闲时间。这一点从流水线的时空图中可以清楚地看到。因此,流水线的效率 E 也就显著提高了。

在计算一条实际流水线的效率时,往往还要考虑流水线各段所使用的设备量不相等,或者流水线各段的价格不相等的情况。在上面给出的所有流水线时空图中,都默认每一个功能段的设备量或设备的价格都是相等的,因此,在纵坐标上每一个功能段都占有一个相等的长度单位。对于可能出现的每一个功能段的设备量或功能段的价格不等的情况,应该根据各个功能段所用的设备量或设备价格在流水线总设备中所占的比例,分别赋予不同的“权”值 p_i 。在绘制流水线时空图时,要根据各功能段权值 p_i 的大小,确定每个功能段在纵坐标上所占的长度,在这种情况下流水线的效率为:

$$E = \frac{n \text{ 个任务占用的加权时空区}}{k \text{ 个流水段的总的加权时空区}}$$

即为:

$$E = \frac{n \cdot \sum_{i=1}^k p_i \cdot \Delta t_i}{\sum_{i=1}^k p_i \left[\sum_{i=1}^k p_i \cdot \Delta t_i + (n-1) \cdot \max(\Delta t_1, \Delta t_2, \dots, \Delta t_n) \right]} \quad (5.40)$$

其中, $p_i < k$, 且 $\sum_{i=1}^k p_i = k$ 。

上面给出了计算流水线吞吐率、加速比和效率的许多个公式,在实际分析一条流水线的性能时,要特别注意这些公式所适用的场合。其中,计算吞吐率的(5.21)、计算加速比的(5.30)和计算效率的(5.34)是适合于各种流水线的。其它公式主要是用于单功能、线性流水线的,而且要求输入的任务是连续的。对于多功能流水线、非线性流水线,或者虽然是线性、单功能流水线,但是输入的任务不是连续的情况,可以结合流水线时空图,并采用通用公式来计算流水线的吞吐率、加速比和效率。在下面有一节中,将专门举例介绍这类流水线的性能分析方法。

5.2.3.4 流水线最佳段数的选择

从上面的分析中可以清楚地看到,增加流水线的段数,流水线的吞吐率和加速比都能提高。但是,由于在每一个功能段的输出端都必须设置一个锁存器(或称缓冲寄存器、闸门寄存器等),因此,当流水线的段数增多时,锁存器的总的延迟时间也将增加,甚至有可能出现锁存器的总延迟时间超过流水线本身的延迟时间。另外,增加锁存器数量,必然要增加流水线的价格。所以,在设计流水线时,要综合各方面的因素,根据最佳性能价格比的要求来选择流水线的最佳段数。

假设在非流水线的机器上采用顺序执行方式完成一个任务所需要的时间为 t , 那么,在同等速度的有 k 段流水线的机器上执行一个任务需要的时间为: $t/k + d$, 其中 d 为锁存器的延迟时间。这样,流水线的最大吞吐率可以表示为: $P = \frac{1}{t/k + d}$ 。流水线的总价格粗略地估计为: $C = a + bk$, 其中, a 为所有功能段本身的总价格, b 为每个锁存器的价格。A. G. Larson把流水线的性能价格比 PCR 定义为:

$$PCR = \frac{P}{C} = \frac{1}{t/k + d} \cdot \frac{1}{a + bk} \quad (5.41)$$

可以通过对自变量 k 求导,得到性能价格比PCR的极值。由于大于零的极值只有一个,因此,这个极值就是最大值。如图 5.41 所示,当性能价格比PCR取得最大值时,它所对应的流水线的段数就是最佳段数 k_0 :

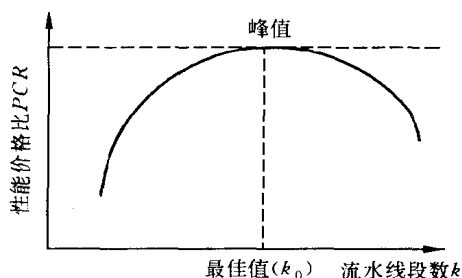


图 5.41 流水线的最佳段数

$$k_0 = \sqrt{\frac{t \cdot a}{d \cdot b}} \quad (5.42)$$

式中的 t 为流水线的总的延迟时间。

从(5.42)关系式中可以很清楚地看到,流水线的最佳段数与流水线的延迟时间 t 和流水线本身的价格 a 的平方根成正比,而与锁存器的延迟时间 d 和锁存器的价格 b 的平方根成反比。

在设计一条流水线的时候,可以根据(5.42)关系式,在流水线的总延迟时间 t 一定的情况下,通过调整流水线本身的价格 a ,锁存器的延迟时间 d 和锁存器的价格 b 来选取最佳的流水线段数 k_0 。

目前,一般处理机中的流水线段数在2段至10段之间,极少有超过15段的流水线。一般把8段或超过8段的流水线称为超流水线,采用8段以上流水线的处理机称为超流水线处理机。

5.2.3.5 流水线性能分析举例

对于单功能、线性流水线,输入任务是连续的情况,可以通过上面给出的有关公式直接计算流水线的吞吐率、加速比和效率。本节通过具体例子介绍在单功能、线性流水线,输入任务是不连续的情况下,以及多功能、线性流水线,输入任务也是不连续的情况下,如何计算流水线的吞吐率、加速比和效率。关于非线性流水线的性能分析方法将在下一节中结合非线性流水线的调度问题一起介绍。

例 5.1 单功能、线性流水线,输入任务是不连续的情况,计算流水线的吞吐率、加速比和效率。

用图 5.25 所示的一条4段浮点加法器流水线计算8个浮点数的和:

$$Z = A + B + C + D + E + F + G + H \quad (5.43)$$

由于存在数据相关,要在 $A+B$ 的运算结果在第4个时钟周期末尾产生之后,在第5个时钟周期才能继续开始做加 C 的运算。这样,在每两个加法运算之间,每个功能部件都要空闲3个时钟周期。这时候,实际上与不采用流水线的顺序执行方式完全一样。

把(5.43)作一个简单的变换,得到:

$$Z = [(A + B) + (C + D)] + [(E + F) + (G + H)] \quad (5.44)$$

小括号内的4个加法操作之间,由于没有数据相关,可以连续输入到流水线中。只要前两个加法的结果出来之后,第一个中括号内的加法就可以开始进行。8个浮点数求和的流水线时空图如图 5.42 所示。

从流水线的时空图中可以很清楚地看到,7个浮点加法共用了15个时钟周期。假设每一个功能段的延迟时间均相等,都为 Δt ,则有 $T_k = 15\Delta t, n = 7$ 。那么,流水线的吞吐率

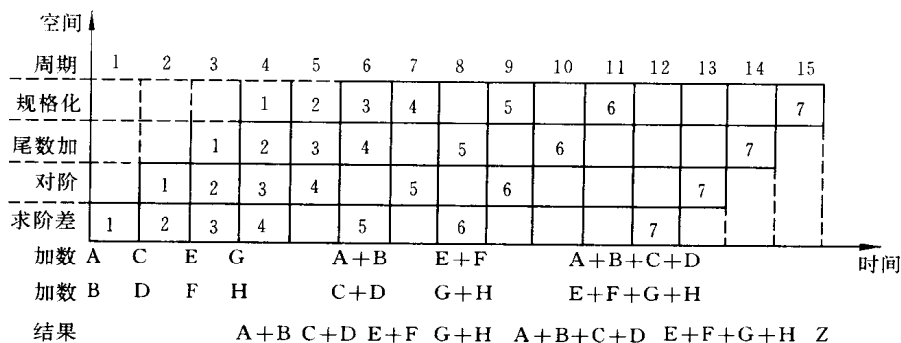


图 5.42 用一条 4 段浮点加法器流水线求 8 个数之和的流水线时空图

TP 为:

$$TP = \frac{n}{T_k} = \frac{7}{15 \cdot \Delta t} = 0.47 \frac{1}{\Delta t}$$

流水线的加速比 S 为:

$$S = \frac{T_0}{T_k} = \frac{4 \times 7 \cdot \Delta t}{15 \cdot \Delta t} = 1.87$$

流水线的效率 E 为:

$$E = \frac{T_0}{k \cdot T_k} = \frac{4 \times 7 \cdot \Delta t}{4 \times 15 \cdot \Delta t} = 0.47$$

例 5.2 多功能、线性流水线,输入任务是不连续的情况,计算流水线的吞吐率、加速比和效率。

用图 5.32 所示的 TI-ASC 计算机的多功能静态流水线计算两个向量的点积:

$$Z = AB + CD + EF + GH \quad (5.45)$$

为了尽量减少数据相关性,充分发挥流水线的作用,计算的顺序应该是先做 4 个乘法: AB 、 CD 、 EF 和 GH ,然后做两个加法 $AB+CD$ 和 $EF+GH$,最后求总的结果 Z 。流水线的时空图如图 5.43 所示。

从流水线时空图中看到,用 20 个时钟周期完成了 7 个运算。当每一个功能段的延迟时间都为 Δt 时,有: $T_k = 20\Delta t$, $n = 7$ 。流水线的吞吐率 TP 为:

$$TP = \frac{n}{T_k} = \frac{7}{20 \cdot \Delta t} = 0.35 \frac{1}{\Delta t}$$

如果采用顺序执行方式,完成一次乘法要用 4 个 Δt ,完成一次加法要用 6 个 Δt ,则完成全部运算要用:

$$T_0 = 4 \times 4\Delta t + 3 \times 6\Delta t = 34\Delta t$$

则流水线的加速比 S 为:

$$S = \frac{T_0}{T_k} = \frac{34 \cdot \Delta t}{20 \cdot \Delta t} = 1.70$$

整个流水线共有 8 段,流水线效率 E 为:

$$E = \frac{T_0}{k \cdot T_k} = \frac{34 \cdot \Delta t}{8 \times 20 \cdot \Delta t} = 0.21$$

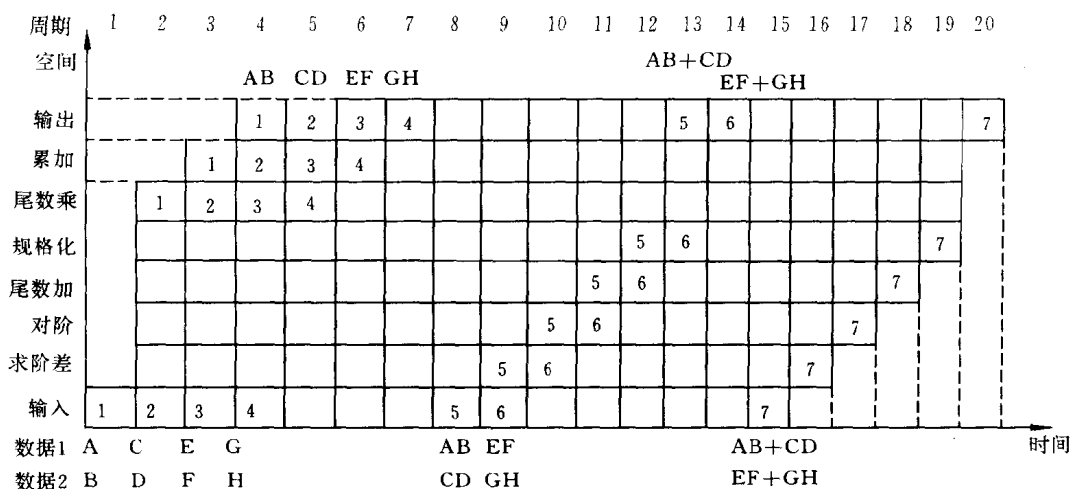


图 5.43 用 TI-ASC 多功能静态流水线求两个向量点积的流水线时空图

整个流水线的效率很低,其原因主要有如下四个。一是多功能流水线在做某一种运算时,总有一些功能段是空闲的;二是静态流水线必须等待前一种运算全部排出流水线之后,才能重新进行连接;三是题目本身存在有数据相关,当发生数据相关时,必须等待前一个运算结果产生之后,下一个运算才能开始;四是流水线有装入与排空部分,当输入到流水线中的任务不多时,装入与排空部分所占的比例比较大。

5.2.4 非线性流水线的调度技术

在线性流水线中,由于每一个任务在流水线的每一功能段中都流过一次,而且仅流过一次,因此,可以在每一个时钟周期向流水线输入一个新任务。在流水线充满之后,每一时钟周期从流水线的输出端产生一个结果。所以,线性流水线的调度非常简单。然而,在非线性流水线中,由于存在有反馈回路,当一个任务在流水线中流过时,在同一个功能段中可能要经过多次。因此,就不能每一个时钟周期向流水线输入一个新任务,否则会在同一个时刻有几个任务争用同一个功能段的情况。这种情况称为功能部件冲突,或流水线冲突。

为了避免流水线发生冲突,一般采用延迟输入新任务的方法。那么,在非线性流水线的输入端,究竟每间隔多少个时钟周期向流水线输入一个新任务才能使流水线的各个功能段都不发生冲突,这就是非线性流水线的调度问题。当然,在一般情况下,这个间隔的时钟周期数应该愈小愈好。在许多非线性流水线中,间隔的周期数往往不是一个常数,而是一串周期变化的数字。因此,非线性流水线调度的任务是要找出一个最小的循环周期,按照这周期向流水线输入新任务,流水线的各个功能段都不会发生冲突,而且流水线的吞吐率和效率最高。

以下,首先介绍非线性流水线的表示方法,然后分析非线性流水线中的冲突情况,并且介绍无冲突调度方法,最后是非线性流水线的优化调度方法。

是就对应有多条非线性流水线的连接图。

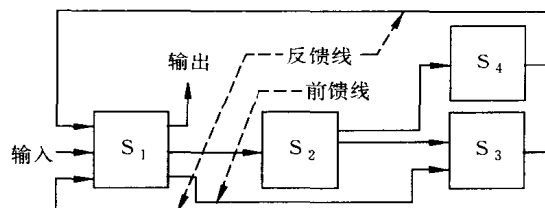


图 5.45 与图 5.44(b)对应的另一种非线性流水线的连接图

同样，一个非线性流水线的连接图也可能对应有多张预约表。例如，图 5.46 就是图 5.44(a)所示流水线连接图的另一张预约表。造成这种情况的原因是：在非线性流水线的有些功能段可能有多个输出端，也有些功能段可能有多个输入端，这些输出端与输入端之间连接的先后次序就形成了多张预约表。

时 间 功能段	1	2	3	4	5	6	7
S_1	×				×		×
S_2		×					
S_3			×			×	
S_4				×			

图 5.46 与图 5.44(a)对应的另一张预约表

5.2.4.2 非线性流水线的冲突

向一条非线性流水线的输入端连续输入两个任务之间的时间间隔称为非线性流水线的启动距离(initiation interval)或等待时间(latency)。启动距离通常用时钟周期数来表示，它是一个正整数。

当以某一个启动距离向一条非线性流水线连续输入任务时，可能在某一个功能段，或某几个功能段中发生有几个任务同时争用同一个功能段的情况，这种情况就是非线性流水线中的冲突(collision)。如图 5.44 所示的一条非线性流水线，当启动距离为 3 时，有关功能段的冲突情况如图 5.47 所示。功能段 S_1 在时钟周期 4 有两个任务争用，在时钟周期 7、10、13、…，有三个任务争用。功能段 S_2 在时钟周期 5、8、11、…，有两个任务争用。

时 间 功能段	1	2	3	4	5	6	7	8	9	10	11	...
S_1	\times_1			$\times_1 \times_2$			$\times_1 \times_2 \times_3$			$\times_2 \times_3 \times_4$...
S_2		\times_1			$\times_1 \times_2$			$\times_2 \times_3$			$\times_3 \times_4$...
S_3		\times_1			\times_2	\times_1		\times_3	\times_2		\times_4	...
S_4			\times_1			\times_2			\times_3			...

图 5.47 启动距离为 3 的流水线冲突情况

同样,图 5.44 所示的非线性流水线,当启动距离为 2 时,在有些功能段中也发生冲突,冲突情况如图 5.48 所示。功能段 S_1 在时钟周期 7、9、11、...,有两个任务争用,功能段 S_3 在时钟周期 6、8、10、...,也有两个任务争用。

时间 功能段	1	2	3	4	5	6	7	8	9	10	11	...
S_1	\times_1		\times_2	\times_1	\times_3	\times_2	$\times_1 \times_4$	\times_3	$\times_2 \times_5$	\times_4	$\times_3 \times_6$...
S_2		\times_1		\times_2	\times_1	\times_3	\times_2	\times_4	\times_3	\times_5	\times_4	...
S_3		\times_1		\times_2		$\times_1 \times_3$		$\times_2 \times_4$		$\times_3 \times_5$...
S_4			\times_1		\times_2		\times_3		\times_4		\times_5	...

图 5.48 启动距离为 2 的流水线冲突情况

引起非线性流水线功能段冲突的启动距离称为禁止启动距离。如图 5.44 所示的非线性流水线,启动距离 2 和启动距离 3 都是禁止启动距离。

有些启动距离在非线性流水线的功能段,在任何时间都不会发生冲突。如图 5.44 的非线性流水线,当启动距离为 5 时的预约表如图 5.49 所示。从这张预约表中可以看出,任何一个功能段在任何时钟周期都不发生冲突。

时间 功能段	1	2	3	4	5	6	7	8	9	10	11	...
S_1	\times_1			\times_1		\times_2	\times_1		\times_2		\times_3	...
S_2		\times_1			\times_1		\times_2			\times_2		...
S_3		\times_1				\times_1	\times_2				\times_2	...
S_4			\times_1					\times_2				...

←
→
 启动周期

←
→
 重复启动周期

图 5.49 启动距离为 5 时的流水线预约表

在非线性流水线中,不发生冲突的启动距离不一定仅仅是一个常数,在一般情况下是一个循环数列。例如,图 5.44 所示的非线性流水线当启动距离为 1、7、1、7、... 交替循环时,任何一个功能段在任何时钟周期也都不发生冲突,如图 5.50 所示。这种使非线性流水线的任何一个功能段在任何时钟周期都不发生冲突的循环数列称为非线性流水线的启动循环。图 5.50 中的启动循环记作(1,7)。图 5.49 中的不发生冲突的启动距离也可以认为是一个循环数列,因此,也可以记作(5)。这种只有一个启动距离的启动循环又称为恒定循环。

要正确地调度一条非线性流水线,首先要找出流水线的所有禁止启动距离。把一条非线性流水线的所有禁止启动距离组合在一起就形成一个数列,通常把这个数列称为非线性流水线的禁止向量。

由预约表得到禁止向量的方法很简单,只要把预约表的每一行中任意两个“ \times ”之间的距离都计算出来,去掉重复的,由这种数组成的一个数列就是这条非线性流水线的禁止向量。例如,对于图 5.44(b)所示的预约表,第 1 行的第 1 列与第 4 列的两个“ \times ”之间的

距离为 3, 第 1 列与第 7 列的两个“×”之间的距离为 6, 第 4 列与第 7 列的两个“×”之间的距离也为 3; 第 2 行的第 2 列与第 5 列的两个“×”之间的距离也为 3; 第 3 行的第 2 列与第 6 列的两个“×”之间的距离为 4。因此, 图 5.44 所示非线性流水线的禁止向量为 (3, 4, 6)。

时间 功能段	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
S_1	× ₁	× ₂		× ₁	× ₂		× ₁	× ₂	× ₃	× ₄		× ₃	× ₄		× ₃	× ₄	...
S_2		× ₁	× ₂		× ₁	× ₂				× ₃	× ₄		× ₃	× ₄			...
S_3		× ₁	× ₂			× ₁	× ₂			× ₃	× ₄			× ₃	× ₄		...
S_4			× ₁	× ₂							× ₃	× ₄					...

图 5.50 启动距离为 (1, 7) 循环时的流水线预约表

把一个启动循环内的所有启动距离相加再除以这个启动循环内的启动距离个数就得到这个启动循环的平均启动距离。例如, 启动循环 (1, 7) 的平均启动距离是 4。而恒定循环 (5) 的平均启动距离就是它本身的启动距离 5。

5.2.4.3 无冲突调度方法

非线性流水线无冲突调度的主要目标是要找出具有最小平均启动距离的启动循环, 按照这样的启动循环向非线性流水线的输入端输入任务, 流水线的工作速度最快, 而且所有功能段在任何时间都没有冲突。下面将系统地介绍非线性流水线的无冲突调度方法, 这些理论最早是由 E·S·Davidson 及其学生们于 1971 年提出来的。

根据一张预约表的禁止向量可以很容易得到冲突向量。冲突向量用一个 m 位的二进制数表示, 其中 m 是禁止向量中的最大值。对于一张 k 列的预约表, 有 $m \leq k - 1$ 。一般的禁止向量用 $C = (C_m C_{m-1} \dots C_2 C_1)$ 来表示。如果 i 在禁止向量中, 则 $C_i = 1$, 否则 $C_i = 0$ 。其中, C_m 一定为 1, 因为 m 必定在禁止向量中。例如, 对于图 5.44(b) 所示预约表, 冲突向量 $C = (101100)$ 。

由冲突向量可以构造一张状态图。把上面得到的冲突向量 C 作为初始冲突向量送入一个 m 位逻辑右移移位器, 当从移位器移出的位为 0 时, 用移位器中的值与初始冲突向量作“按位或”运算, 得到一个新的冲突向量; 若移位器移出的位为 1, 不作任何处理; 移位器继续右移, 如此重复, 每当移位器移出的位为 0 时, 都把移位器中的值与初始冲突向量作按位或运算, 得到一个新的冲突向量, 这样的操作共进行 m 次。对于中间形成的每一个新的冲突向量, 也要按照这一方法进行处理。在初始冲突向量和所有的新形成的冲突向量之间用带箭头的线连接, 表示各种状态之间的转换关系。当新形成的冲突向量出现重复时可以合并到一起。

由图 5.44 所示非线性流水线构成的状态图如图 5.51 所示。初始冲突向量为 101100, 它逻辑右移 3、4、6 位时, 由于移出去的位是 1, 表示用这些启动距离向流水线输入新任务, 在流水线中会发生功能段的冲突, 因此, 在状态图中不作任何处理。当逻辑右移

的位数为 1、2、5 和大于等于 7 时,移出去的位是 0,表示用这些启动距离向流水线输入新任务时,在流水线的各个功能段中都不会发生冲突。但是,这里所说的不发生功能段冲突,只能保证输入一个新任务时,在流水线的各个功能段都不发生冲突。在输入这个新任务之后,流水线又将产生新的冲突向量,这时,要根据新产生的冲突向量来判断流水线发生冲突的情况。

在图 5.51 中,初始冲突向量 101100 右移一位之后得到 010110,两数作“按位或”运算 $101100 \vee 010110 = 111110$,111110 是一个新的冲突向量。在初始冲突向量 101100 与新形成的冲突向量 111110 之间用一条带箭头的线连接,并在带箭头的线旁边注上数字 1,表示初始冲突向量是经过一次右移产生的冲突向量 111110。同样,初始冲突向量经过两次右移后再与原来的初始冲突向量进行“按位或”运算 $001011 \vee 101100 = 101111$,在初始冲突向量 101100 与新形成的冲突向量 101111 之间用一条带箭头的线连接,并在带箭头的线旁边注上数字 2。初始冲突向量经过 5 次右移后再与原来的初始冲突向量进行“按位或”运算 $000001 \vee 101100 = 101101$,在初始冲突向量 101100 与新形成的冲突向量 101101 之间用一条带箭头的线连接,并在带箭头的线旁边注上数字 5。初始冲突向量 101100 经过 7 次或大于 7 次的逻辑右移后再与原来的初始冲突向量进行“按位或”运算,结果必然还是原来的初始冲突向量本身,在图 5.51 中用一条从初始冲突向量出发又指向它自己的弧形线表示,并在弧形线内标注数字 7^* 。“ 7^* ”的意思是大于等于 7。至此,初始冲突向量已经全部处理完成。对于新形成的 3 个冲突向量 111110、101111 和 101101 也要采用与初始冲突向量相同的方法进行处理。冲突向量 111110 经过 1 次右移后与初始冲突向量进行“按位或”运算 $011111 \vee 101100 = 111111$,在冲突向量 111110 与再次形成的新冲突向量 111111 之间用一条带箭头的线连接,并在带箭头的线旁边注上数字 1。冲突向量 111111 经过 7 次或大于 7 次的右移后与初始冲突向量进行“按位或”运算,又回到初始冲突向量,因此,在图 5.51 中用一条弧形线从冲突向量 111111 连向初始冲突向量 101100,并在弧形线旁标注数字 7^* 。按照同样的方法处理另外两个新形成的冲突向量 101111 和 101101,结果如图 5.51 所示。

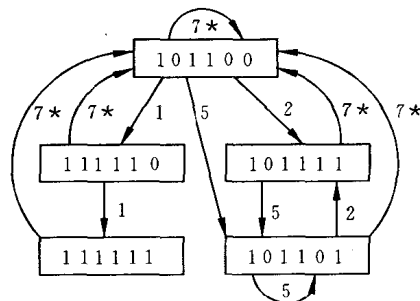


图 5.51 非线性流水线的状态图

当初始冲突向量确定之后,状态图就是唯一的,因此,与一张预约表相对应,只有唯一的一个状态图。但是,由于不同的预约表可能产生相同的初始冲突向量,因而,不同的预约表也可能有相同的状态图。所以,从预约表可以画出状态图,但从状态图不能得到预约表。

从状态图中可以看到,当启动距离大于或等于 $m+1$ 时,流水线的任何一个功能段在任何时钟周期都不会发生冲突;但是,这时流水线的吞吐率、加速比和效率都将很差;因此,非线性流水线调度的任务就是要找出平均启动距离小于 m 的启动循环,按照这个启动循环向流水线的输入端连续输入新任务,流水线的任何一个功能段在任何时钟周期都不会发生冲突。

在状态图中可以找到很多不发生功能段冲突的启动循环。实际上,这样的启动循环有

无穷多个。例如,从图 5.51 中可以找到的启动循环有(2,7)、(2,5,7)、(2,5,2,7)、(2,5)、(2,5,5,7)、…。因为非线性流水线调度的主要目标要找出平均启动距离最小的启动循环,因此,在这些无穷多个启动循环中,一般只要找出简单循环即可。

所谓简单循环是指在状态图中各种冲突向量只经过一次的启动循环。在一个状态图中,简单循环的个数一般是有限的。例如,在上述启动循环中,(2,7)、(2,5,7)、(2,5)是简单循环,而(2,5,2,7)和(2,5,5,7)不是简单循环,因为(2,5,2,7)两次经过冲突向量(101111), (2,5,5,7)两次经过冲突向量(101101)。

状态图 5.51 中的所有简单循环均包括在表 5.1 中,共有 8 个简单循环。在这个表中,还分别计算出了这些简单循环的平均启动距离。

表 5.1 所有简单循环的平均启动距离

简单循环	平均启动距离
(1,7)	4
(1,1,7)	3
(2,7)	4.5
(2,5)	3.5
(2,5,7)	7
(5,7)	6
(5)	5
(7)	7
(5,2,7)	4.7

从表 5.1 中可以很容易地找到平均启动距离最小的启动循环,这样的启动循环被称为最小启动循环。在状态图 5.51 中的最小启动循环是(1,1,7),其平均启动距离是 $(1+1+7)/3=3$ 。当非线性流水线按照这个最小启动循环工作时,在向流水线输入第 1 个任务之后,要相隔 1 个时钟周期再向流水线输入第 2 个任务,然后再相隔 1 个时钟周期,向流水线输入第 3 个任务,再相隔 7 个时钟周期向流水线输入第 4 个任务;如此重复,连续向流水线输入新任务,则流水线的各个功能段在任何时钟周期都不会发生冲突。图 5.44 所示非线性流水线在按照最小启动循环(1,1,7)工作时,它的流水线预约表如图 5.52 所示。

时间 功能段	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
S ₁	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	...
S ₂		×	×	×	×	×	×				×	×	×	×	×	...
S ₃		×	×	×		×	×	×			×	×	×		×	...
S ₄			×	×	×							×	×	×		...

启动周期
 重复启动周期

图 5.52 最小启动循环(1,1,7)的流水线预约表

图 5.52 是一种不等间隔的流水线调度方法,它的控制比较复杂。为了简化流水线的控制逻辑,也可以采用相等间隔的调度方法,即用恒定循环来启动流水线。从表 5.1 中可

以看出,启动距离最小的恒定循环是(5)。按照恒定循环(5)工作的流水线预约表如图5.49所示。

5.2.4.4 优化调度方法

从图5.52中可以看到,当采用最小启动循环启动非线性流水线时,流水线中的许多功能段还有空闲。反映在预约表中还有空白的格子。因此,按照上一节介绍的调度方法得到的最小启动循环,实际上并不能使非线性流水线充分发挥效率。下面将介绍非线性流水线的优化调度方法,按照这种方法调度非线性流水线,流水线的工作效率最高,即流水线的吞吐率、加速比和效率最好。

L. E. Shar 于 1972 年提出了流水线最小平均启动距离的限制范围。对于一条静态可重构的流水线,通过预约表可以得到其最小平均启动距离的范围。

1. 最小平均启动距离的下限是预约表中任意一行里“×”的最多个数。实际上也就是同一个任务通过流水线中任意一个功能段的最多次数。

2. 最小平均启动距离应小于或等于状态图中任意一个简单循环的平均启动距离。这一点已经在上一节中被使用。

3. 最小平均启动距离的上限是冲突向量中 1 的个数再加上 1。这个限制在许多情况下是相当宽松的。

1992 年, L. E. Shar 又证明了上述限制范围。

实际上,最有用的是上述第 1 条。预约表中“×”最多的一行所对应的功能段一定是整个流水线的瓶颈功能段。要使整个流水线充分发挥效率,瓶颈功能段必须不间断工作,不能有空闲。因此,非线性流水线调度的关键是充分使用瓶颈功能段,只要让瓶颈功能段不空闲,则非线性流水线的吞吐率、加速比和效率必然是最好的。按照这种方法调度流水线,流水线的平均启动距离也一定是最小的。

采用预留算法来调度非线性流水线,可以达到最优调度。具体方法如下:

1. 确定流水线的最小平均启动距离。最小平均启动距离等于预约表中任意一行中“×”的最大个数,或者是同一个任务通过流水线中任意一个功能段的最多次数。

2. 确定最小启动循环。相对于同一个最小平均启动距离可能有多个最小启动循环,其中有一个,而且只有一个启动距离都相等的恒定循环。为了简化流水线的控制逻辑,在一般情况下,就选择这个恒定循环作为最小启动循环。

3. 结合流水线预约表和连接图,采用预留算法,通过插入非计算延迟功能段实现最小启动循环。

以下,结合图 5.44 的例子,具体说明调度方法。用预留算法调度的预约表、连接图和状态图如图 5.53 所示。

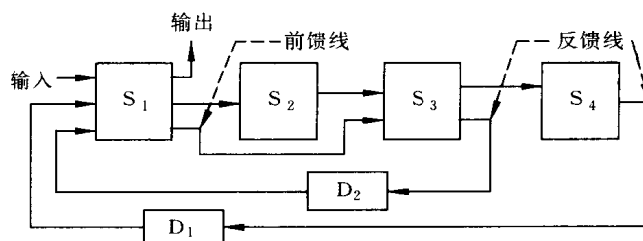
从图 5.53(a)所示的预约表中很容易看到,只有功能段 S_1 所对应的行有 3 个“×”,因此,功能段 S_1 就是整个非线性流水线的瓶颈功能段。于是,可以确定流水线的最小平均启动距离为 3。

与最小平均启动距离 3 相对应的最小启动循环可能有 3 个,它们是(3)、(1,5)和(2,4)。其中(3)是一种恒定循环,由于每次启动距离都相等,因此,流水线的控制比较简

时间		1	2	3	4	5	6	7	8	9
功能段	S ₁	×			①→×			①→②→×		
	S ₂		×			①→×				
	S ₃		×				①→×			
	S ₄			×						
延迟	D ₁				×					
	D ₂								×	

注：①表示由D₁延迟一个时钟周期，②表示由D₂延迟一个时钟周期

(a) 有非计算延迟的预约表



(b) 有非计算延迟的流水线连接图

图 5.53 有非计算延迟的非线性流水线

单。下面，先以最小启动循环(3)为例，说明预留算法的调度方法。

从预约表中看，任意一行中凡是与第 1 个“×”的距离为 3 的倍数的时钟周期都要预留出来。于是，功能段 S₁ 所在行的第 2 个“×”要从时钟周期 4 向后延迟到时钟周期 5，同样，功能段 S₂ 所在行的第 2 个“×”也要从时钟周期 5 向后延迟到时钟周期 6。另外，由于功能段 S₃ 所在行的第 2 个“×”的输入来自功能段 S₂ 的第 2 个“×”的输出，因此，功能段 S₃ 所在行的第 2 个“×”也要向后延迟一个时钟周期，从时钟周期 6 延迟到时钟周期 7。按照这种方法调度之后，用最小启动循环(3)启动流水线，预约表中的第 1 个“×”和第 2 个“×”都不会发生冲突。由于功能段 S₁ 所在行共有 3 个“×”，因此，还要检查第 3 个“×”是否有冲突。第 2 个“×”现在在时钟周期 5，因此 5+3 这个时钟周期必须预留，因此，功能段 S₁ 所在行的第 3 个“×”还要向后延迟一个时钟周期，从上一次延迟到的时钟周期 8 再向后延迟一个时钟周期到时钟周期 9。经过这样调度后的预约表如图 5.53(a)所示，与这张预约表相对应的流水线连接图是图 5.53(b)；它与图 5.44(a)相比，增加了两个非计算延迟功能段 D₁ 和 D₂，这两个功能段在流水线中的作用只是延迟一个时钟周期，没有其它功能。

如果采用不等启动距离的调度方案，如采用最小启动循环(1,5)或(2,4)，调度方法与上面所述相同。

与图 5.53 相对应的流水线状态图如图 5.54 所示。由于初始冲突向量中“0”的个数比较多，因此，状态图的连接关系比较复杂。从状态图中可以很容易地看到，流水线的最小启动循环是(3)。

按照最小启动循环(3)工作的流水线预约表如图 5.55 所示。每间隔 3 个时钟周期，可以向流水线输入一个新任务。

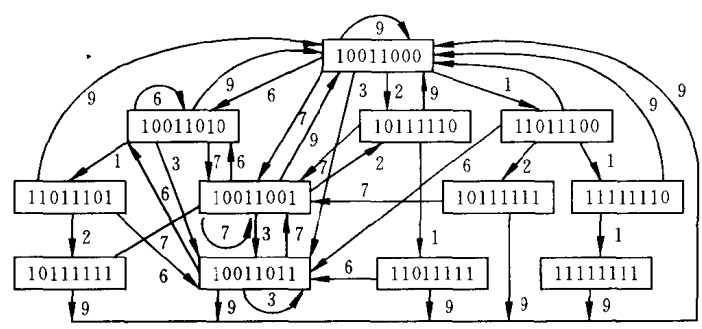


图 5.54 有非计算延迟的流水线状态图

从图 5.55 中可以看到,按照最小启动循环(3)工作,流水线的任何一个功能段在任何
一个时钟周期都不发生冲突。另外,从时钟周期 7 开始,瓶颈功能段 S_1 就一直忙碌,没有
一个时钟周期是空闲的,因此,采用这种调度方法能够使线性流水线的吞吐率、加速比和
效率达到最优。

时间		1	2	3	4	5	6	7	8	9	10	11	12	...
功能段	S ₁	×			×	×		×	×	×	×	×	×	...
	S ₂		×			×	×		×	×		×	×	...
	S ₃			×				×	×		×	×		...
	S ₄				×		×			×			×	...
延迟	D ₁				×			×			×			...
	D ₂								×			×		...

←一个周期→

←重复周期→

←重复周期→

←重复周期→

图 5.55 按照最小启动循环(3)工作的流水线预约表

以上介绍的非线性流水线调度技术主要适用于单功能非线性流水线和静态多功能非
线性流水线。对于动态多功能非线性流水线的调度,只需要把每种功能的预约表重叠在一
起,采用上面介绍的单功能非线性流水线的基本调度方法,不难解决动态多功能非线性流
水线的调度问题。

5.2.5 局部相关

在前面介绍先行控制技术时,曾经把处理机中的相关分为两大类,一类是数据相关,
另一类是控制相关。在流水线处理机中,由于同时执行的指令条数很多,发生相关的可能
性及其造成的影响将更加严重。

按照对程序执行过程可能造成的影响来划分,可以把相关划分为
局部相关和全局相关两类。如图 5.56 所示,如果程序内有一个两
路的条件分支操作指令,它把程序划分为三个部分 B_0 、 B_1 和 B_2 ,在
每一部分内部不再有分支操作指令,通常把这样的—个部分称为—
个基本块(basic block)。在图 5.56 中共有三个基本块。在同一个基
本块内部的相关称为局部相关(local correlation),在基本块之间的

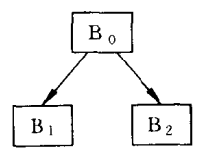


图 5.56 全局相关与局部相关

相关称为全局相关(global correlation)。引起全局相关的除了条件分支操作之外,还有中断等。

一般来说,局部相关对程序执行过程的影响相对比较小,它仅影响到相关指令前后的一条或几条指令的执行,而全局相关造成的影响比局部相关要大得多,它影响到整个程序的执行方向。

要使流水线充分发挥效率,不发生或少发生“断流”情况,需要硬件和软件两方面的共同努力。在软件方面,编译器生成的目标程序要能够适合流水线结构,要尽量把有数据相关和控制相关的指令安排得相隔远一些,把相同的操作尽量安排在一起等。在硬件方面,要解决好存储系统的频带平衡问题,能够为流水线提供足够的指令和数据,除此之外,处理好流水线的局部相关和全局相关也非常重要。在本节及下节中,将分别介绍流水线的局部相关和全局相关发生的原因及其解决方法。

在指令流水线中,局部相关主要有三种,“先写后读”数据相关,简称“写读”相关,记作“WR”或“RAW”;“先读后写”数据相关,简称“读写”相关,记作“RW”或“WAR”;“写-写”相关,记作“WW”或“WAW”。在下面的三小节中,主要分析这三种相关发生的原因和避免的办法。

5.2.5.1 顺序流动与乱序流动

在一般情况下,一串连续任务在流水线中是一个接一个地在各个功能段中间流过的。从流水线的输出端看,任务流出流水线的顺序与输入端的任务流入顺序完全相同,这种控制方式称为顺序流动方式。如图 5.57 所示的一条 6 段指令流水线,在正常情况下,6 个功能段 S_0 、 S_1 、 \dots 、 S_5 分别同时执行 k 、 $k+1$ 、 \dots 、 $k+5$ 共 6 条指令,如图 5.58 中的最下面一行。

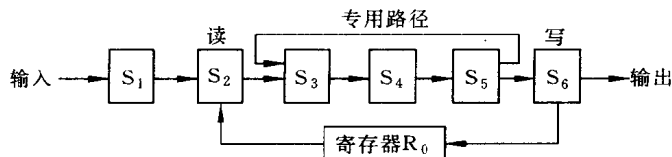


图 5.57 一条 6 段指令流水线

在图 5.57 中,功能段 S_2 要读操作数,功能段 S_6 写运算结果。现在把如下一段程序输入到这条流水线中:

```

k:      R0 ← (R1)
k+1:    ...
k+2:    R2 ← (R0) + (R3)
k+3:    ...
k+4:    ...
k+5:    ...
...

```

其中,指令 k 与指令 $k+2$ 之间有“先写后读”数据相关。在时钟周期 t_i ,指令 $k+2$ 在功能

段 S_2 中要读操作数;但是,由于指令 k 还没有到达功能段 S_6 ;因此,指令 $k+2$ 无法继续执行,要在功能段 S_2 中等待。如图 5.58 所示,后续的指令 $k+4$ 、 $k+5$ 、 \dots 等也不能进入流水线。在以后的 t_{i+1} 、 t_{i+2} 、 t_{i+3} 、 \dots 时钟周期,功能段 S_3 、 S_4 、 S_5 将逐渐空闲。在时钟周期 t_{i+3} ,指令 k 执行完成,把运算结果写到寄存器 $R0$ 中。从时钟周期 t_{i+4} 开始,指令又可以在流水线中继续往前流动。然而,三个“空闲段”要在流水线的后四个功能段中流过,直至全部流出流水线。

从上面的分析及图 5.58 中可以看出,采用顺序流动方式,在程序中有“先写后读”数据相关时,流水线就会“断流”,有些功能段要出现“空闲”,这将降低流水线的吞吐率和效率。当然,采用顺序流动方式,流水线的控制逻辑比较简单。

	时钟周期 t						
t_{i+4}	$k+4$	$k+3$	$k+2$	空闲	空闲	空闲	
t_{i+3}	$k+3$	$k+2$	空闲	空闲	空闲	$k+1$	
t_{i+2}	$k+3$	$k+2$	空闲	空闲	$k+1$	k	
t_{i+1}	$k+3$	$k+2$	空闲	$k+1$	k	$k-1$	
t_i	$k+3$	$k+2$	$k+1$	k	$k-1$	$k-2$	
正常流动	$k+5$	$k+4$	$k+3$	$k+2$	$k+1$	k	功能段 S
功能段	S_1	S_2	S_3	S_4	S_5	S_6	

图 5.58 顺序流动方式

为了充分发挥流水线的效率,在发生数据相关时,要允许没有数据相关的后续指令进入相关指令所占用的功能段执行,并超越相关的指令继续往前流动,这种控制方式称为乱序流动方式。从一条指令流水线的输出端看,指令流出流水线的顺序与输入端指令输入流水线的顺序是不一样的。在流水线的输入端,指令是按照它的地址从小到大顺序进入流水线的;但在流水线的输出端,指令执行完成的顺序可能是混乱的;因此,把流水线的这种控制方式称为乱序(out of order)流动方式,有些资料上也称为错序流动方式、无序流动方式或异步流动方式等。

仍以上面一小段程序为例。在时钟周期 t_{i+1} ,指令 $k+2$ 在功能段 S_2 中要读操作数,即指令 k 与指令 $k+2$ 之间有“先写后读”数据相关。如图 5.59 所示,如果后续的指令 $k+3$ 、 $k+4$ 、 \dots 等与指令 $k+2$ 没有数据相关,在 t_{i+1} 、 t_{i+2} 、 t_{i+3} 时钟周期,可以把指令 $k+2$ 一直保存在功能段 S_2 中,而让后续的指令 $k+3$ 、 $k+4$ 、 $k+5$ 进入功能段 S_2 执行,并超越指令 $k+2$ 进入以后的功能段。在时钟周期 t_{i+3} ,指令 k 执行完成,把运算结果写到寄存器 $R0$ 中。在时钟周期 t_{i+4} ,功能段 S_2 可以执行指令 $k+2$,于是,指令 $k+2$ 可以跟在指令 $k+5$ 之后继续往前流动。因此,从流水线的输出端看,指令流出流水线的顺序与输入端指令进入流水线的顺序是不一样的。

实际上,造成流水线乱序流动方式的原因不仅仅是“先写后读”数据相关,有些指令执行的时间短或经过的功能段比较少,它们要越过执行时间长或经过功能段多的指令向前流动。

	时钟周期 <i>t</i>					
t_{i+5}	$k+8$	$(k+7)$ $k+6$	$k+2$	$k+5$	$k+4$	$k+3$
t_{i+4}	$k+7$	$(k+6)$ $k+2$	$k+5$	$k+4$	$k+3$	$k+1$
t_{i+3}	$k+6$	$k+5$ $(k+2)$	$k+4$	$k+3$	$k+1$	k
t_{i+2}	$k+5$	$k+4$ $(k+2)$	$k+3$	$k+1$	k	$k-1$
t_{i+1}	$k+4$	$k+3$ $(k+2)$	$k+1$	k	$k-1$	$k-2$
t_i	$k+3$	$(k+2)$ $k+1$	k	$k-1$	$k-1$	$k-3$
正常流动	$k+5$	$k+4$	$k+3$	$k+2$	$k+1$	k
功能段	S_1	S_2	S_3	S_4	S_5	S_6

图 5.59 乱序流动方式

在乱序流动方式中,除了有“先写后读”数据相关之外,还可能发生“先读后写”数据相关和“写-写”数据相关。例如,把下面一小段程序输入到动态多功能流水线,或者多条单功能流水线中,当指令 $k+2$ 进入读操作数功能段时,发现指令 k 与指令 $k+2$ 之间有“先写后读”数据相关,指令 $k+2$ 不能继续执行。后续的指令 $k+3$ 、 $k+4$ 、 \dots 等可以进入读操作数功能段先执行,并超越指令 $k+2$ 进入以后的功能段。由于指令 k 和指令 $k+2$ 是乘法指令,执行的时间比较长,指令 $k+3$ 有可能超越指令 k 先到达写功能段,把它的执行结果写到了寄存器 R3 中。于是,当指令 k 在写功能段执行完成,解除了它与指令 $k+2$ 之间的“先写后读”数据相关,指令 $k+2$ 要读寄存器 R3;但是,寄存器 R3 已经被指令 $k+3$ 修改过了,这就是“先读后写”数据相关。同样,如果指令 $k+4$ 超越指令 $k+2$ 先到达写功能段,寄存器 R2 中的最终结果必然是错误的,这就是“写-写”数据相关。

```

k:      R0=R1×R4
k+1:    R6=R5+1
k+2:    R2=R0×R3
k+3:    R3=R4-1
k+4:    R2=R5
k+5:    ...

```

测试“先写后读”数据相关的方法是在流水线的读操作数功能段设置一个相联比较器,在指令读操作数之前,把源操作数地址与已经在流水线中的从读操作数功能段到写结果功能段之间的所有指令的目标地址进行比较,如果发现有一个地址是相等的,则表明发生了“先写后读”数据相关。类似地,测试“先读后写”数据相关和“写-写”数据相关的方法是在流水线的写结果功能段设置相联比较器,把自己的目标操作数地址分别与已经进入流水线的指令序号比自己小的源操作数地址和目标操作数地址进行比较,如果发现与某一条指令的源操作数地址相等,则说明发生了“先读后写”数据相关,如果发现与某一条指

令的目标操作数地址相等,则说明发生了“写-写”数据相关。

当在流水线中已经测试到有“先写后读”、“先读后写”或“写-写”三种数据相关时,必须采取适当的措施来处理这些相关。处理数据相关的方法有很多种,在下面的三小节中,分别介绍三种常用的方法。

5.2.5.2 数据相关及其避免方法

在流水线中的数据相关,在逻辑设计中经常称为“冒险”(hazard)或“竞争”(competition)等,可以用图 5.60 来形象化地描述它们。

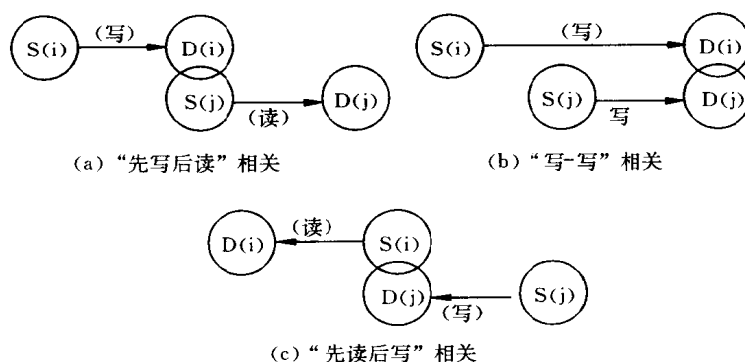


图 5.60 在指令流水线中可能出现的数据相关(指令 i 先于指令 j)

在图 5.60 中,按照程序原定的指令执行次序,指令 i 应该先于指令 j 执行。如果采用顺序流动方式,则可能发生如图 5.60(a)所示的“先写后读”相关。如果由于某种原因,指令的实际执行次序违反了程序原定的次序,在流水线中的指令 j 超越了指令 i,则除了可能发生“先写后读”相关之外,还可能发生如图 5.60(c)所示的“先读后写”或图 5.60(b)所示的“写-写”相关。

图 5.60 中的三种数据相关可以用下列关系式来表示。

$$\begin{aligned}
 &\text{对于“先写后读”相关} && D(i) \cap S(j) \neq \emptyset \\
 &\text{对于“先读后写”相关} && S(i) \cap D(j) \neq \emptyset \\
 &\text{对于“写-写”相关} && D(i) \cap D(j) \neq \emptyset
 \end{aligned} \tag{5.46}$$

如果(5.46)中的三个关系式之一满足,就可能发生数据相关;但是,这只是发生相关的必要条件,而不是充分条件;也就是说,当可能发生数据相关时,可以采取适当的措施来避免相关。在处理机执行程序的过程中,只有避免了数据相关,程序执行结果的语义才是正确的。

在流水线中避免发生数据相关的方法可以分为两大类,其中一类是延迟执行,另一类是建立专用路径。当流水线的功能段比较多,或者在一个处理机中有多条流水线时,需要的专用路径的条数很多,专用路径的控制非常复杂;因此,出现了多种设置专用路径的专门方法,如采用分散控制的公共数据总线法(又称为 Tomasulo 算法),采用集中控制的 CDC 记分牌法等。

延迟执行是避免数据相关最简单的方法。例如,在图 5.57 中,当检测到指令 k 与指令

$k+2$ 有“先写后读”数据相关时,延迟指令 $k+2$ 的执行,让流水线的功能段 S_3 、 S_4 、 S_5 空转,直到相关被解除之后,指令 $k+2$ 才继续在流水线中执行。采用延迟执行方法避免数据相关的流水线工作时序如图 5.58 所示。这种方法的优点是流水线的控制简单,缺点是流水线的吞吐率和效率低。

在一条单功能线性流水线中,建立专用路径的方法如图 5.57 所示。通过一条从功能段 S_5 到功能段 S_3 专用路径,当指令 k 与指令 $k+2$ 有“先写后读”数据相关时,功能段 S_5 的输出可以通过这条专用路径直接送到功能段 S_3 的输入。在采用顺序流动方式时,建立有这种专用数据路径的流水线工作时序如图 5.61 所示。与没有建立专用数据路径的图 5.58 相比,指令 $k+2$ 可以提前两个时钟周期执行,从而流水线的吞吐率和效率有比较大的提高。由于在一般程序中,这类数据相关发生的概率很大,因此,在流水线中建立专用数据路径已经成为高性能处理机普遍采用的方法。

	时钟周期 t						
t_{i+3}	$k+5$	$k+4$	$k+3$	$k+2$	空闲	$k+1$	功能段 S
t_{i+2}	$k+4$	$k+3$	$k+2$	空闲	$k+1$	k	
t_{i+1}	$k+3$	$k+2$	空闲	$k+1$	k	$k-1$	
t_i	$k+3$	$k+2$	$k+1$	k	$k-1$	$k-2$	
功能段	S_1	S_2	S_3	S_4	S_5	S_6	

图 5.61 建立有专用路径的顺序流动方式

通过建立专用路径来避免数据相关的基本原理是数据重定向。下面,首先介绍常用的数据重定向的一般原理和基本方法,在此基础上,再介绍一种经典的避免数据相关的 Tomasulo 算法。

5.2.5.3 数据重定向

在单条流水线中,建立专用路径的方法主要有两种。

(1) 对于“先写后读”数据相关,如图 5.62(a)所示。先 $A \rightarrow B$,然后 $B \rightarrow C$;等效于 $A \rightarrow B \rightarrow C$ 。新增加了一条从 A 到 C 的专用路径,撤消原来从 B 到 C 的路径。经过这样的改变之后,就可以避免“先写后读”数据相关,或者缩短发生“先写后读”数据相关的时间。

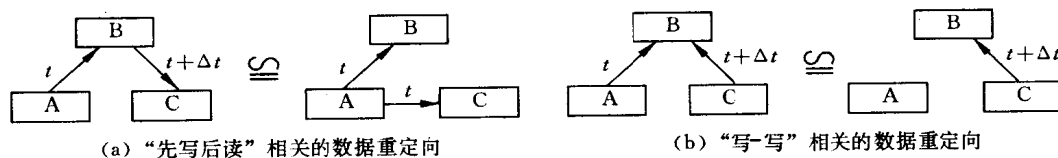


图 5.62 数据重定向原理

(2) 对于“写-写”数据相关,如图 5.62(b)所示。先 $A \rightarrow B$,然后 $C \rightarrow B$;经过数据重定向,等效于 $C \rightarrow B$,撤消原来从 A 到 B 的路径。经过这样的改变之后,就可以完全避免发生“写-写”数据相关。

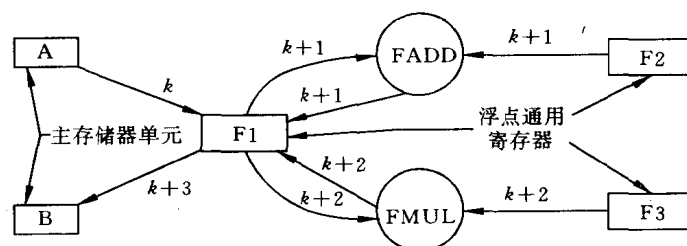
另外,“先读后写”数据相关一般发生在多功能流水线或有多条流水线的处理机中,在只有一个写功能段的单条流水线中一般不会发生“先读后写”数据相关。在有多条流水线的超标量处理机中,解决“先读后写”数据相关的方法通常是设置一个缓冲寄存器,只要把发生相关的源存储单元中的内容复制到这个缓冲寄存器中,“先写后读”数据相关就可以避免。

下面,通过一个具体的例子来说明在程序执行过程中,是如何通过数据重定向来避免数据相关的。一个简单的程序如下:

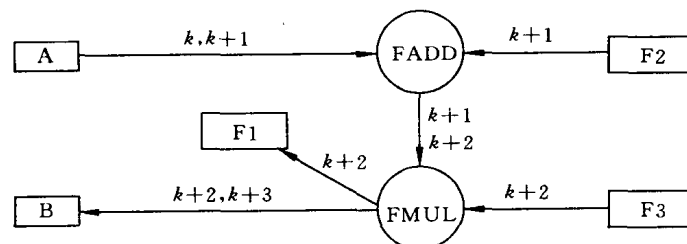
```
k:      LOAD   F1, A
k+1:    FADD   F1, F2
k+2:    FMUL   F1, F3
k+3:    STORE  F1, B
```

程序中的 F1、F2、F3 是浮点通用寄存器, A 和 B 是主存储器的单元。程序执行过程中数据流程如图 5.63(a) 所示,图中的长方形表示数据存储单元,圆形表示运算部件,带有箭头的线表示数据传送路径。

根据上面介绍的数据重定向的基本原理,为了处理“先写后读”数据相关,专门设置了 $A \rightarrow \text{FADD}$ 、 $\text{FADD} \rightarrow \text{FMUL}$ 、 $\text{FMUL} \rightarrow B$ 三条专用路径,同时,撤消了 $F1 \rightarrow \text{FADD}$ 和 $F1 \rightarrow \text{FMUL}$ 的路径。另外,为了处理“写-写”数据相关,撤消了 $A \rightarrow F1$ 、 $\text{FADD} \rightarrow F1$ 的数据传送路径。数据流程如图 5.63(b) 所示。



(a) 原始数据流程



(b) 重定向之后的数据流程

图 5.63 数据重定向原理

在流水线处理机中,经常有多条流水线,每条流水线有多个功能段,能够同时执行多条指令。如果为每一种可能出现的数据相关都建立专门的数据路径,那么,不仅相关路径

的数量很多,而且控制逻辑也相当复杂。为了处理流水线中可能出现的多种数据相关,人们提出了多种解决数据相关的算法。

5.2.5.4 Tomasulo 动态指令调度算法

Tomasulo 算法是由 R. M. Tomasulo 于 1967 年首先提出的,并最早在 IBM 360/91 处理机的浮点处理部件中被采用。在有的资料上,Tomasulo 算法又被称为公共数据总线(CDB;common data bus)法,或令牌法等。它采用乱序流动方式来提高流水线的吞吐率和效率,并通过分散控制的办法处理数据相关。

在 IBM 360/91 处理机的浮点处理部件中,有一个浮点加法器和一个浮点乘/除法器,如图 5.64 所示。浮点加法器为两段流水线,在它的输入端有三个保存站 A1、A2 和 A3,浮点乘/除法器为六段流水线,在它的输入端有两个保存站 M1 和 M2。每个保存站的组成如图中所示,它采用随机访问方式工作,由保存站中的控制部件控制。当任意一个保存站中的两个源操作数都到齐之后,如果对应的浮点操作部件是空闲的,则可以把两个操作数立即送入浮点操作部件中执行。这种工作方式很像将要在最后一章中介绍的数据流计算机中的数据驱动方式。

IBM 360/91 处理机的浮点处理部件采用先行控制方式。在图 5.64 中的浮点先行操作站、浮点先行读数站、浮点后行写数站的作用和工作原理与前面已经介绍过的先行控制方式相同。在浮点先行操作站中存放的是经过指令分析部件预处理之后的“寄存器-寄存器”型指令,这类指令中的源操作数可能来自浮点通用寄存器,也可能来自浮点先行读数站,运算结果送入公共数据总线 CDB,并通过 CDB 送入浮点通用寄存器、浮点加法器的保存站或浮点乘/除法器的保存站等;最终运算结果一般送入浮点后行写数站,由浮点后行写数站负责写到主存储器中去。

仍然以上一节中的一小段程序为例。如果在串行执行的处理机中,必须等待上一条指令执行完后,下一条指令才能开始执行。由于浮点加法器和浮点乘法器可以同时工作,如果采用乱序流动方式,在上面的这四条指令中可能发生“先写后读”和“写-写”两种数据相关。

指令 k 的执行结果,把一个浮点操作数从主存储器中读出,送入浮点先行读数站的(FLB1)中,并在执行指令 $k+1$ 时通过(FLB1)直接送入到浮点加法器的保存站 A1,而不是送到浮点通用寄存器 F1 中,这就相当于建立了一条从主存储器到浮点加法器的专用数据路径。

在执行指令 $k+1$ 时,把浮点通用寄存器(F2)中的内容通过 FLR 总线送入浮点加法器的保存站 A1 中,同时把浮点先行读数站(FLB1)中的操作数通过另一条 FLB 总线也送入浮点加法器的保存站 A1 中,并启动浮点加法器开始工作。因为浮点加法器的运算结果要送入 F1,因此,要把浮点加法器保存站 A1 的站号“10”装入浮点通用寄存器 F1 的站号中;另外,要把 F1 的“忙位”标志置“1”,表示浮点通用寄存器 F1 中的内容目前不能读出作为源操作数来使用。

由于浮点加法器要执行两个时钟周期才能出结果,当执行指令 $k+2$ 时,虽然可以把浮点通用寄存器(F3)中的内容通过 FLR 总线送入浮点乘/除法器的保存站 M1 中;但是,

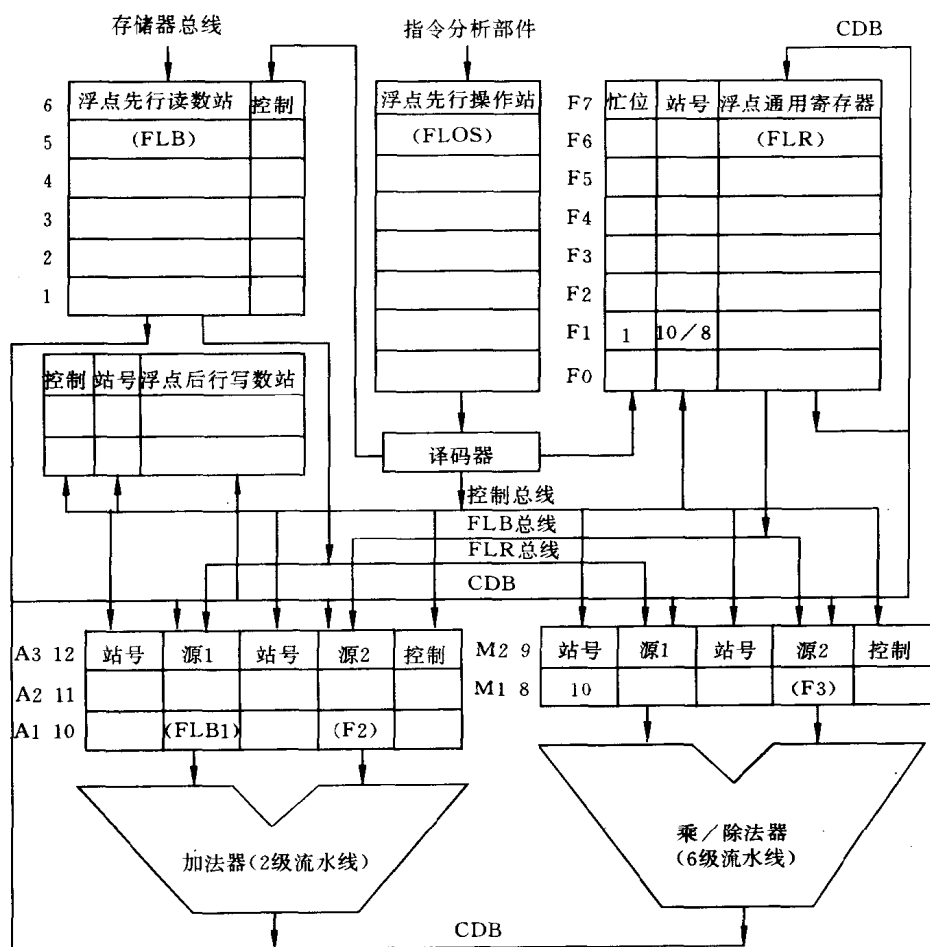


图 5.64 IBM360/91 处理机的浮点执行部件

执行浮点乘法所需要的另一个操作数还不能从浮点通用寄存器 F1 中读出;这时,改为把保存在浮点通用寄存器 F1 中的站号“10”(浮点加法器保存站 A1 的站号)读出来,并通过控制总线送入到浮点乘/除法器的保存站 M1 中,并把 F1 内的站号改为浮点乘/除法器保存站 M1 的站号“8”,表示改从浮点乘/除法器接收运算结果。

在浮点加法执行完成之后,把运算结果和保存站 A1 的站号“10”都送上公共数据总线 CDB。通过与所有站号中保存的内容进行相等比较,找到浮点乘/除法器的保存站 M1;于是,把 CDB 上的数据送入 M1,而不是送入浮点通用寄存器 F1;这相当于建立了一条从浮点加法器到浮点乘/除法器的专用数据路径。

浮点乘/除法器收到 CDB 上的数据之后就可以开始工作。在 6 个时钟周期之后,浮点乘法的执行结果连同保存站 M1 的站号都送上 CDB。通过与所有站号进行相等比较,把 CDB 上的数据送入到浮点通用寄存器 F1 中。

在浮点乘/除法器刚刚开始工作时,指令 $k+3$ 开始执行。由于浮点乘法的最终运算结果还没有出来,因此,就把保存站 M1 的站号“8”装入后行写数站的站号中。当浮点乘法操

作执行结束之后,在把运算结果通过 CDB 送入浮点通用寄存器 F1 的同时,也送入浮点后行写数站 SDB,由 SDB 负责把这个结果写到主存储器的 B 单元中。

实际上,数据在浮点执行部件中的流动过程就象图 6.63(b)那样,与图 5.63(a)的串行执行方式相比,程序的执行速度要快得多。

另一种可以用来动态调度多条流水线的经典方法是 CDC 计分牌法,由 J. E. Thornton 于 1970 年提出。由于这种方法最先在 CDC 6 600 大型计算机系统中被采用,因此,通常称为“CDC 计分牌法”。为了控制数据寄存器与操作部件之间的数据传送,在 CDC 6 600 中设置有一个被称为计分牌(scoreboard)的集中控制部件。在计分牌中保存有与各个操作部件相联系的寄存器中的数据装载情况。当一个操作部件所要求的数据都已经装载完成之后,计分牌就允许操作部件开始运行。在操作执行完成之后,操作部件通知计分牌释放有关资源。计分牌是一个集中控制部件,它记录了数据寄存器和多个操作部件状态的变化情况,可以通过它来消除或减少某些数据相关,加快程序的执行速度。

5.2.6 全局相关

由条件转移或程序中断引起的相关被称为全局相关。在流水线中,全局相关对流水线的吞吐率和效率的影响相对于局部相关要大得多,而且,条件转移指令在一般程序中所占的比例相当大。中断虽然在程序中所占的比例不大,但是,中断发生在程序中的哪一条指令,发生在一条指令执行过程中的哪一个功能段都是不确定的。因此,处理好条件转移和中断引起的全局相关是很重要的。其中的关键问题有两个,一是要确保流水线能够正常工作,二是减少因“断流”引起的吞吐率和效率的下降。

无论是条件码在上一条指令中形成的一般条件转移指令,还是条件码在本条指令中形成的复合条件转移指令,一般情况下都要在转移指令执行到流水线的最后功能段时,转移条件才能建立。因此,在条件转移指令进入流水线之后,到形成转移条件之前,后续指令不能进入流水线。很显然,这会使流水线的吞吐率和效率严重下降。从相关的角度看,条件转移指令或断点指令与后续指令之间存在着一种相关,使它们不能同时进入流水线中执行。这种相关与上面介绍的数据相关或局部相关不同,这是一种控制上的相关,因此,被称为控制相关,或全局相关。

下面,首先分析条件转移对流水线性能的影响,然后分别介绍条件转移和程序中断的处理方法。

5.2.6.1 转移的影响

与前面介绍的先行控制方式相同,在遇到条件转移指令时,为了使流水线不“断流”,通常采用“猜测法”。在条件转移指令之后,选择一个分支方向,让后续指令进入流水线执行。在图 5.65 中,猜测的分支方向是固定,它选择的是转移不成功的方向,当然,也可以选择转移成功的方向。

在一般情况下,猜测转移不成功的方向,其控制逻辑相对比较简单,因为在遇到条件转移指令时,可以让流水线按照原先的顺序继续往前流动。如图 5.65 所示,第 i 条指令是条件转移指令,第 $i+1$ 、 $i+2$ 、 \dots 条指令可以按虚线方向继续进入流水线执行。因为第 i 条

条件转移指令所需要的条件码由第 $i-1$ 条指令给出；在一条有 k 个功能段的流水线中，第 $i-1$ 条指令要等到第 $i+k-2$ 条指令进入流水线时才能形成条件码。如果形成的条件码是对应于转移不成功的，则猜测正确，流水线的吞吐率和效率没有降低，就好像没有条件转移指令一样；相反，如果形成的条件码是对应于转移成功的，则猜测错误，要改为沿实线方向，先作废流水线中已经执行的第 $i+1, i+2, \dots, i+k-2$ 条指令，然后，再从分支点开始执行第 $p, p+1, \dots$ 条指令。在这种情况下，每执行一条条件转移指令，一条 k 段流水线有 $k-2$ 个功能段是浪费的。

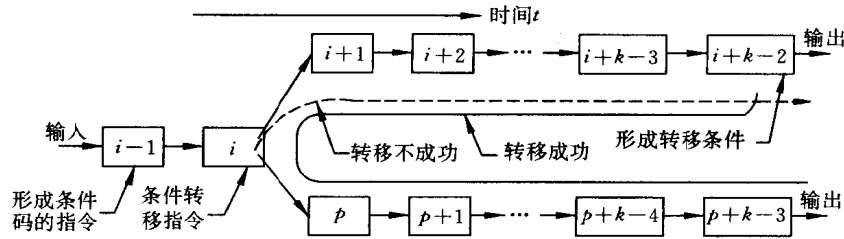


图 5.65 条件转移指令在流水线中的执行过程

当分支方向猜测错误时，不仅流水线中有多个功能段要浪费，更严重的是可能造成程序执行结果发生错误。例如，若第 $i+1$ 条指令是：

$$(R1) + (R2) \rightarrow R1$$

一旦这条指令执行完成，寄存器 $R1$ 中内容就被破坏，整个程序执行的结果也将是错误的。因此，当流水线沿猜测方向执行第 $i+1, i+2, \dots$ 条指令时，一定不能破坏通用寄存器和主存储器中的内容。为做到这一点，目前的处理机有两种做法。一种方法是只进行指令译码和准备好运算所需要的操作数，在转移条件没有形成之前不执行运算；另一种方法是一直执行到运算完成，但不送回运算结果。前一种方法，即使猜测正确时，也可能造成流水线功能段的浪费，但控制逻辑比较简单；后一种方法则要设置一定数量的缓冲寄存器来存放中间运算结果，而且，控制逻辑比较复杂。

下面，具体分析条件转移指令对流水线性能的影响。

对于一条有 k 个功能段的流水线，由于条件转移指令的影响，在最坏情况下，每一次条件转移将造成 $k-1$ 个时钟周期的“断流”。另外，假设条件转移指令在一般程序中所占的比例为 p ，转移成功的概率为 q 。因此，对于一个由 n 条指令组成的程序，在执行这个程序的过程中，由于条件转移需要额外增加的时钟周期数是： $pqn(k-1)\Delta t$ 。包括条件转移指令在内的 n 条指令的总的执行时间是：

$$T_{K-IF} = (n + k - 1)\Delta t + npq(k - 1)\Delta t$$

根据计算流水线吞吐率的一般公式，即前面介绍的(5.21)式，有条件转移影响的流水线的吞吐率为：

$$TP_{IF} = \frac{n}{(n + k - 1)\Delta t + npq(k - 1)\Delta t} \quad (5.47)$$

当 $n \rightarrow \infty$ 时，有条件转移影响的流水线的最大吞吐率为：

$$TP_{MAX-IF} = \frac{1}{(1 + pq(k - 1))\Delta t} \quad (5.48)$$

在没有条件转移指令,即 $p = q = 0$ 的情况下,流水线的最大吞吐率 $TP_{\text{MAX}} = \frac{1}{\Delta t}$,与前面得到的(5.24)式相同。

由于条件转移指令的影响,流水线吞吐率下降的百分比为:

$$D = \frac{TP_{\text{MAX}} - TP_{\text{MAX-IF}}}{TP_{\text{MAX}}} = \frac{pq(k-1)}{1 - pq(k-1)} \quad (5.49)$$

据统计,在一些典型程序中,转移指令所占的比例为 $p = 20\%$,转移成功的概率为 $q = 60\%$ 。对于一条有 8 个功能段的指令流水线,由于条件转移指令的影响,流水线的最大吞吐率要下降:

$$D_8 = \frac{0.20 \times 0.60 \times (8-1)}{1 + 0.20 \times 0.60 \times (8-1)} = 46\%$$

同样,如果指令流水线的功能段数为 10,由于条件转移指令的影响,流水线的最大吞吐率将下降到原来的一半以下:

$$D_{10} = \frac{0.20 \times 0.60 \times (10-1)}{1 + 0.20 \times 0.60 \times (10-1)} = 52\%$$

从上面的分析中可以看到,条件转移指令对流水线的影响是非常大的,必须采取措施来减少这种影响。可能采取的措施有如下几种:

1. 延迟转移技术和指令取消技术:这两种技术在本书第二章的“RISC 的关键技术”一节中介绍过了。延迟转移技术是在遇到转移指令时,依靠编译器把一条或几条没有数据相关和控制相关的指令调度到转移指令的后面。当被调度的指令执行完成之后,转移指令的有效目标地址也已经计算出来了。

延迟转移技术一般只用于单流水线标量处理机中,而且流水线的级数不能太多。因为流水线的级数愈多,需要调度到转移指令后面的没有数据相关和控制相关的指令条数也要愈多。据统计,编译器调度一条指令成功的概率在 90% 以上,而调度两条指令成功的概率只有 40% 左右。当没有合适的指令可调度时,编译器只能插入空操作。

SUN 公司的 SPARC 处理机、HP 公司的 HPPA 处理机及 SGI 公司的一部分 MIPS 处理机采用了延迟转移技术。在 HPPA 处理机中还采用了指令取消技术。

2. 静态转移预测技术:主要内容已经在本章的第一部分的“先行控制技术”中介绍了。所谓静态转移预测是指在处理机的硬件和软件设计完成之后,转移预测的方向就已经确定了,或者预测为转移不成功的方向,或是预测为转移成功的方向,在程序实际执行过程中,转移预测的方向不能改变。

静态转移预测可以只用软件实现,也可用硬件来实现,还可以在转移的两个方向上都预取指令。采用最后面一种方法时,硬件要设置一个转移目标缓冲栈及相应的控制逻辑,当然,转移造成的损失也可以减少些。

TI 公司的 SuperSPARC 处理机采用了静态转移预测技术,而且设置有转移目标缓冲栈,在两个方向上都预取指令。

3. 动态转移预测技术:它根据近期转移是否成功的历史记录来预测下一次转移的方向。所有的动态转移预测方法都能够随程序的执行过程动态地改变转移的预测方向。在下一小节中将具体介绍动态转移预测技术。

4. 提前形成条件码:这种方法能够很有效地缓解因条件转移造成的流水线“断流”问题,下面将要具体介绍这种方法。

5.2.6.2 动态转移预测技术

动态转移预测技术的关键是要解决好两个问题,一是如何记录转移历史信息,另一个是如何根据所记录的转移历史信息预测转移的方向。

记录转移历史信息的方法通常有三种。第一种是把最近一次或几次转移是否成功的信息记录在转移指令中,第二种是用一个小容量的高速缓冲栈保存条件转移指令的转移目标地址,第三种是用 Cache 保存转移目标地址之后的 n 条指令。下面,根据三种不同的记录转移历史信息的方法介绍三类动态转移预测技术。

方法一,在指令 Cache 中记录转移历史信息

在指令 Cache 中专门设置一个字段,称为“转移历史表”。在执行转移指令时,把转移成功或不成功的信息记录在这个“转移历史表”中。可以只用一个二进制位来记录最近一次转移是否成功的信息,也可以用多个二进制位来记录最近几次转移是否成功的信息。当下次再执行到这条指令时,转移预测逻辑根据“转移历史表”中记录的信息预测转移成功或不成功。

当采用只记录最近一次转移是否成功的历史信息时,每一条指令的“转移历史表”只需要一个二进制位,它的状态转换图如图 5.66(a)所示。每个圆圈表示一种状态,圆圈中的“T”或“N”表示最近一次执行这条转移指令时,实际转移成功或不成功的信息,这个信息也就是“转移历史表”中所记录的内容。在执行条件转移指令时,如果“转移历史表”中记录的内容是“T”,则转移预测逻辑预测转移成功,并按照转移成功的方向取指令并分析指令;如果记录的是“N”,则按照转移不成功的方向继续取下一条指令并分析指令。在转移条件实际形成之后,如果与预测的转移方向相同,则预测正确,流水线没有任何“断流”损失;如果与预测的转移方向不同,则要作废已经预取和分析的指令,并从另一个方向取指令和分析指令。

无论预测的转移方向与指令实际执行结果的转移方向是否相同,都要用本条转移指令实际转移是否成功的信息来修改“转移历史表”。修改的方法可以按照图 5.66(a)所示状态转换图进行,图中带有箭头的线表示状态转换的方向,线旁边的 T 表示指令实际执行结果为转移成功,而 N 表示指令实际执行结果为转移不成功。当采用只记录最近一次转移是否成功的历史信息时,“转移历史表”的修改方法很简单。如果“转移历史表”中原来记录的是“T”,本次转移成功,则“转移历史表”中的内容继续保持“T”;如果原来记录的是“T”,本次转移不成功,则“转移历史表”中的内容要修改为“N”。同样,如果“转移历史表”中原来记录的是“N”,本次转移不成功,则“转移历史表”中的内容继续保持“N”;如果原来记录的是“N”,本次转移成功,则“转移历史表”中的内容要修改为“T”。

为了取得更好的转移预测效果,通常需要记录最近多次转移是否成功的历史信息,这时,“转移历史表”也需要有多个二进制位。转移历史信息记录得愈多,状态转换关系也就愈复杂。记录最近两次转移是否成功的历史信息的状态转换关系如图 5.66(b)所示。与图 5.66(a)类似,每个圆圈表示一种状态,圆圈中的上一行表示最近两次执行这条转移指令

时,实际转移成功或不成功的信息,这个信息也是“转移历史表”中所记录的内容。圆圈中的下一行表示在当前这种状态下,本次执行这条转移指令时预测的转移方向。

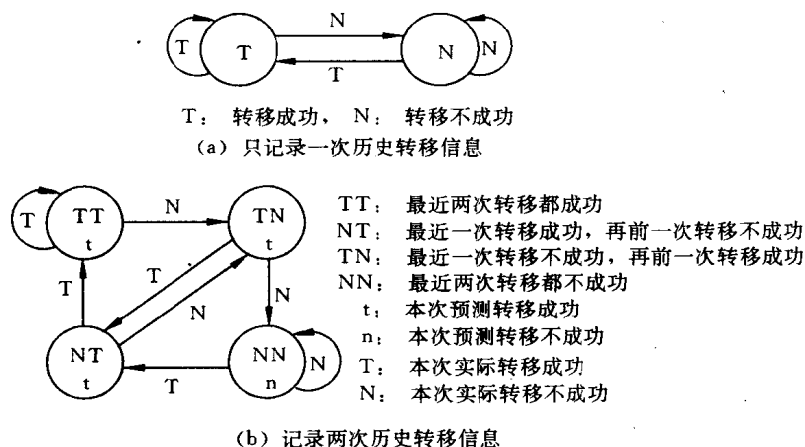


图 5.66 记录历史转移信息的状态转换图

在图 5.66(b)中,对于左上角的状态,圆圈中的“TT”表示历史上最近两次执行这条转移指令时转移都成功,本次又执行这条转移指令,预测转移成功。如果实际执行结果转移又成功,则“转移历史表”中所记录的内容仍保持“TT”;如果实际执行结果转移不成功,则要转向右上角的状态“TN”,把“转移历史表”中记录的内容修改为“TN”。同样,对于右上角的状态,圆圈中的“TN”表示历史上最近一次执行这条转移指令时转移不成功,而再上一次执行这条指令时转移是成功的,本次又执行这条转移指令,预测转移成功。如果实际执行结果转移成功,则转向左下角的状态“NT”,把“转移历史表”中记录的内容修改为“NT”;如果实际执行结果转移不成功,则转向右下角的状态“NN”,把“转移历史表”中记录的内容修改为“NN”。对于另外两种状态,可以结合图 5.66(b)自己进行分析。

实际上,在图 5.66(b)中,修改“转移历史表”的规则是采用了一种向左移位的方法,右边新移入的就是本次执行这条转移指令时转移是否成功的信息;而转移预测是偏向于转移成功的,只有历史上最近两次执行这条转移指令时转移都没有成功,本次才预测转移不成功,对于另外三种情况都预测转移成功。

“转移历史表”的修改规则和转移预测规则可以多种多样,“转移历史表”除了记录转移是否成功的历史信息之外,也可以记录转移预测是否成功的信息。同样,预测本次转移方向时,也可以用最近预测是否成功的信息作为依据。

当一条转移指令第一次被执行时,它还没有转移历史的记录,即“转移历史表”还是空白的。可以有两种做法,一种是在“转移历史表”中预置转移历史信息。例如,可以预置历史上转移都成功。那么,在第一次执行这条转移指令时,必然按照转移成功的方向进行预测。这种预置转移历史信息的做法,转移预测逻辑相对比较简单。另一种方法是根据指令本身的偏移字段的符号来预测转移的方向。如果偏移字段的符号是负的,则预测转移成功,否则预测转移不成功。当偏移字段的符号是负时,表示要向后转移,通常是转移到一个循环程序的开始,因此,转移成功的概率比较大。

采用在指令 Cache 中记录转移历史信息的方法,其优点是不必专门设置转移缓冲栈,所记录的转移历史信息比较少。它的缺点是需要有专门的指令 Cache 和专门的指令 Cache 控制逻辑。

DEC 公司的 Alpha 21064 超标量超流水线处理机就采用了这种转移预测方法,在它的一级指令 Cache 中有一个专门的“转移历史表”字段。

方法二,转移目标地址缓冲栈

用一个小容量的高速缓冲栈保存最近执行的 k 条转移指令的“转移历史表”和转移目标地址,如图 5.67 所示。“转移指令地址”字段采用全相联方式访问。当程序中执行到一条转移指令时,把当前指令地址与转移目标缓冲栈中的所有转移指令地址进行比较。如果发现有所相等的,则根据同一行中的“转移历史表”所记录的历史信息预测本次转移的方向,同时用转移目标地址预取指令。在实际转移条件形成之后,可以根据某一种规则修改“转移历史表”

I_0	T_0	P_0
I_1	T_1	P_1
I_2	T_2	P_2
\vdots	\vdots	\vdots
I_{k-1}	T_{k-1}	P_{k-1}
转移指令地址	转移历史表	转移目标地址

图 5.67 转移目标地址缓冲栈

方法三,转移目标指令缓冲栈

当转移指令在指令分析部件中译码时,转移不成功方向上的指令已经被预取到先行指令缓冲栈中,或者已经存放在指令 Cache 中。为了能够在转移成功方向上也预取一部分指令,可以把图 5.67 中的转移目标地址部分改为存放转移目标地址之后的 n 条指令。设置转移目标指令缓冲栈的转移预测方法如图 5.68 所示,它的工作原理、预测转移方向的规则和修改“转移历史表”的方法与方法二相同。

A_0	T_0	$I_{0,0}$	$I_{0,1}$...	$I_{0,n-1}$
A_1	T_1	$I_{1,0}$	$I_{1,1}$...	$I_{1,n-1}$
A_2	T_2	$I_{2,0}$	$I_{2,1}$...	$I_{2,n-1}$
\vdots	\vdots				
A_{k-1}	T_{k-1}	$I_{k-1,0}$	$I_{k-1,1}$...	$I_{k-1,n-1}$
转移指令地址	转移历史表	转移目标地址之后的 n 条指令			

图 5.68 转移目标指令缓冲栈

动态转移预测的方法有很多种,预测的准确性除了与程序本身的特性有关之外,还主要与记录的历史信息的复杂程度有关。一般来说,记录的历史信息愈复杂,其预测的准确性也愈高,当然,所需要的硬件代价也愈大。

5.2.6.3 提前形成条件码

从上一节的分析中可以看出,条件转移指令造成流水线吞吐率下降的主要原因是条

件码形成得太晚。因此,尽早产生条件码对减少流水线吞吐率和效率的损失非常有效。那么,条件码能否提前产生呢?

对于一般条件转移指令,转移条件码是由上一条运算型指令产生的。对于大多数情况,可以在运算实际开始之前或者在运算中间就能产生条件码,不必等到运算完成之后。例如,乘法或除法指令,两个源操作数符号相同,结果为正,符号相反,结果为负;两个源操作数中有一个为“0”,则乘积为“0”;被除数为“0”,则商为“0”,除数为“0”,则除法结果溢出。因此,只要比较两个操作数的符号或阶码就能够确定运算结果的正负号、是否为“0”、是否溢出等。对于加法或减法运算,相同符号的两个数相加,结果的符号与其中一个加数的符号相同;两个不同符号的操作数相减,结果的符号与被减数的符号相同;符号不同的两个操作数相加,或者符号相同的两个操作数相减,结果的符号与两个操作数中绝对值较大的那个操作数的符号相同;加法和减法的溢出、结果是否为“0”等也可以通过一个简单的比较器提前产生。因此,在绝大多数情况下,只要在运算部件的入口处设置一个比较器,通过比较两个操作数的符号或者阶码就能够提前形成结果的正负号、是否为“0”、是否溢出等条件码。如果能够在—个时钟之内就产生条件码,则正好可以提供给下一条条件转移指令使用,这样,流水线就不会“断流”。例如,Amdahl 470V/6 计算机在运算部件的入口处设置一个 LOCK 部件来预判条件码。

有一种用来控制循环次数的计数转移指令在条件转移指令中占有很大的比例,而且这种指令要随着循环程序反复执行很多次,因此,在许多流水线处理机中对这种条件转移要做特殊的处理。例如,有下面一段循环程序:

```

        LOAD  R1, NUM ; 把循环次数初值装入到寄存器 R1 中
LOOP:   ...           ; 循环体开始
        ...
        ...           ; 循环体结束
        DEC   R1       ; 循环次数减“1”
        BNE   LOOP     ; 如果循环次数未到,继续执行循环体,否则结束循环
        HELT                ; 程序结束
NUM:    n               ; 循环次数

```

这段程序中的最后两条指令是一种典型的条件转移指令的组合,条件转移指令所需要的条件码是由前一条减“1”指令产生的。对于这种循环程序中的计数转移指令,在本书中已经介绍过两种处理方法。一种是在第二章的“RISC 关键技术”中介绍的延迟转移技术;另一种是在本章的“先行控制技术”中介绍的转移预测技术。下面,介绍在流水线处理机中经常采用的另一种方法。

首先,要由编译程序在对源程序进行编译时发现这种计数转移指令。当程序中出现这种指令组合时,把“DEC R1”指令提前到循环体前面。这样,上面一段循环程序被编译成如下程序:

```

        LOAD R1, NUM ; 把循环次数初值装入到寄存器 R1 中
LOOP:   LDEC R1      ; 这是一条专门的循环次数减“1”指令
        ...           ; 循环体开始

```

```

...
...           ; 循环体结束
LBNE LOOP    ; 这是一条专门用来测试循环是否结束的指令
HELT        ; 程序结束
NUM:  n      ; 循环次数

```

程序中的“LDEC”和“LBNE”是两条专门用来控制循环次数的指令，它们使用另外一个专门的条件码寄存器 CC_L 。 CC_L 由“LDEC”循环次数减“1”指令设置，由“LBNE”循环出口测试指令使用。而处于这两条指令中间的属于循环体的指令不能破坏 CC_L 条件码寄存器中的内容。

采用专门的循环控制指令和专门的条件码寄存器，能够把产生条件码的运算型指令与使用这个条件码的条件转移指令分离开来。只要循环体长度不是很短，循环体中的所有指令再加上两条循环控制指令能够充满流水线，流水线就不会有“断流”情况发生。这种方法已经在不少的处理机中得到应用。

5.2.6.4 精确断点与不精确断点

当正在处理机中运行的一个程序被某一个中断源中断时，因为在流水线中同时有多条指令在执行，那么，断点究竟在哪一条指令呢？

对于常规输入输出设备申请的输入输出中断服务，其目的是要求处理机暂时停止正在执行的程序，转去执行一段中断服务程序，以完成设备所要求的输入输出任务。对于这种情况，实际上不需要有精确的断点。因此，比较简单的处理方法是：让已经进入流水线的指令都执行完成，断点就是最后进入流水线的那条指令的地址。如图 5.69 中，断点地址是 $i+5$ 。中断服务程序可以紧接在第 $i+5$ 条指令之后进入流水线。

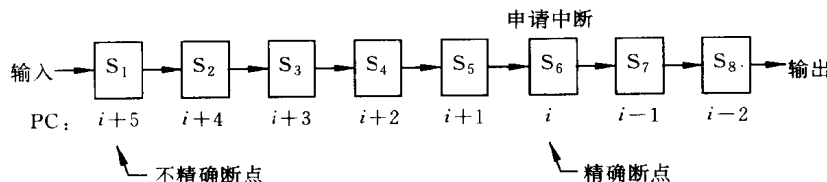


图 5.69 流水线处理机的中断处理

由于常规输入输出设备的中断服务在整个中断处理中占据主要部分，采用不精确断点法，只要能够及时从主存储器中读入中断服务程序，流水线可以不“断流”。而且，采用不精确断点法所需要的硬件比较少，控制逻辑相对比较简单。它的主要缺点是中断响应时间稍长些，一般要增加一条流水线的长度。

对于程序性错误和机器故障等引起的中断，它们出现的概率很低，在整个程序执行过程中所占的比例很小。因此，流水线处理机处理这类中断的出发点不在于如何缩短“断流”时间，关键是要正确保存现场和正确恢复断点。

如图 5.69 中，第 i 条指令执行到功能段 S_6 时出现程序性错误，可能的程序性错误有指令或数据格式错、主存保护、非法操作码、地址越界、各种运算结果溢出、调用管理程序

等,可能的机器故障有信息校验错误(包括电源故障、运算电路误动作、主存储器出错、输入输出设备或控制器出错、机器的其它各种硬件故障等)、虚拟存储器缺页等。这时,第 $i+1, i+2, \dots, i+5$ 条指令都已经进入流水线。如果采用乱序流动方式,则第 $i+1, i+2, \dots$ 条指令可能已经超越第 i 条指令,这种情况下,已经进入流水线中的指令可能更多。

处理因为程序性错误和机器故障等产生的中断,一般情况下可以有两种方法,一种是采用不精确断点(imprecise interrupt)方法,另一种是采用精确断点(precise interrupt)方法。

如果采用不精确断点法,中断处理过程与常规的输入输出设备相同。凡是已经进入流水线的指令都执行完成,断点就是最后进入流水线的那条指令的地址。然而,采用不精确断点法可能会发生如下两个问题。一个问题是程序执行的结果可能出错。例如,有如下两条指令:

```
i:    FADD  R1, R2      ; (R1)+(R2)→R1
i+1: FMUL  R3, R1      ; (R3)×(R1)→R3
```

当第 i 条指令执行到功能段 S_6 时发现浮点加法结果溢出,于是发出中断服务申请。由于采用不精确断点法,要让已经进入流水线的第 $i+1$ 条指令也执行完成。因为第 $i+1$ 条指令使用了不正确的通用寄存器 $R1$ 中的内容作源操作数,因此,浮点乘法的执行结果一般也是不正确的。另一个问题是在程序调试过程中,通常要设置断点,程序员通过查看断点处的中间执行结果来判断程序编写是否正确。如果采用不精确断点法,由于程序不能准确中断在程序员所设置的断点处,因此,在这种流水线处理机上,程序员很难调试程序。在流水线处理机出现的初期,因为采用了这种不精确断点法,曾经有一些程序设计人员批评流水线处理机,而且不喜欢使用流水线处理机。

在流水线的功能段数目不是很多时,只需采取适当措施,例如,判断发生程序性错误的指令与已经进入流水线的其它指令之间是否有数据相关等。只要发生程序性错误的指令不影响流水线中其它指令的执行结果,在这种情况下,采用不精确断点法仍然能够保证程序的执行结果是正确的。在早期生产的一批大型流水线处理机中,有许多采用了不精确断点法,例如,IBM 360/91 计算机等。

近期生产的流水线处理机一般都采用精确断点法。由哪一条指令的程序性错误或故障发出的中断申请,断点就是这条指令的地址。例如,在图 5.69 中,如果第 i 条指令发生了程序性错误或故障,那么,断点就是 i 。为了实现精确断点法,需要把断点以前的指令的执行结果都保存下来。最坏情况发生在指令执行到最后一个功能段时才发生程序性错误或故障。为此,要设置一定数量的后援寄存器,把整个流水线中所有指令的执行结果和现场都保存下来,以便在发生中断时,能够恢复到精确断点。因此,采用精确断点法所需要的硬件代价比较高,控制逻辑也比较复杂。

5.3 超标量处理机与超流水线处理机

在已经掌握了一般流水线标量处理机的基础上,本节主要介绍另外三种高性能的指

令级并行处理机,即超标量处理机(superscalar processor)、超流水线处理机(super-pipelining processor)和超标量超流水线处理机(superpipelining superscalar processor)的基本原理、典型结构和它们的主要性能等。

如果以一台 k 段流水线的普通标量处理机为基准,超标量处理机、超流水线处理机和超标量超流水线处理机的主要性能比较如表 5.2 所示。

表 5.2 四种不同类型处理机的性能比较

机器类型	k 段流水线 基准标量处理机	m 度 超标量处理机	n 度 超流水线处理机	(m, n) 度超标量 超流水线处理机
机器流水线周期	1 个时钟周期	1	$1/n$	$1/n$
同时发射指令条数	1 条	m	1	m
指令发射等待时间	1 个时钟周期	1	$1/n$	$1/n$
指令级并行度 ILP	1	m	n	$m \times n$

在表 5.2 中,基准标量处理机是一台普通的单流水线处理机。为了便于进行比较,把基准标量处理机的机器流水线周期和指令发射等待时间都假设为 1 个时钟周期,同时发射的指令条数为一条,它的指令级并行度 ILP (instruction level parallelism)假设为 1。另外三种指令级并行处理机,即并行度为 m 的超标量处理机,并行度为 n 的超流水线处理机,以及并行度为 (m, n) 的超标量超流水线处理机,它们的性能都相对于基准标量处理机进行比较。

5.3.1 超标量处理机

一般的流水线处理机只有一条指令流水线,一个多功能的操作部件,每个时钟周期“取指令”和“分析”完成一条指令。在许多流水线处理机中,指令流水线的功能段数 $k=4$,它把一条指令的执行过程主要分解为“取指令”、“分析”、“执行”和“写回运算结果”4 个阶段。指令所要执行的功能主要多功能操作部件中,在“执行”这一功能段完成。多数流水线处理机的多功能操作部件采用流水线结构。有的简单指令,只要一个时钟周期就能够在“执行”功能段中完成,而比较复杂的指令往往需要多个时钟周期才能够做完,另外,还有条件转移等的影响。因此,一般流水线标量处理机每个时钟周期平均执行指令的条数小于 1,即它的指令级并行度 $ILP < 1$ 。

另一种流水线处理机虽然也只有一条指令流水线,每个时钟周期“取指令”和“分析”完成一条指令,但是有多个独立的操作部件,例如,定点算术逻辑部件、浮点加减法部件、乘除法部件、取数存数部件等。这种处理机通常称为多操作部件处理机。在多操作部件处理机中,操作部件可以采用流水线结构,也可以不采用流水线结构。如果操作部件不采用流水线结构,虽然有些操作部件的“执行”时间要多于一个时钟周期,但是,由于每个时钟周期可以“取指令”和“分析”完成一条指令,因此,多个操作部件仍然可以同时并行工作,这样可以充分发挥多个操作部件的作用。在一般情况下,多操作部件处理机的指令级并行度 ILP 也小于 1。

以下将要介绍的超标量、超流水线和超标量超流水线三种处理机在一个时钟周期内可以执行完成多条指令,即它们的指令级并行度 ILP 都大于 1。

在目前已经实用的微处理机中,大多数属于超标量处理机。例如,Intel 公司的 i860、i960、Pentium 处理机,Motolora 公司的 MC88110,IBM 公司的 Power 6000,SUN 公司的 SuperSPARC 等都是超标量处理机。SGI 公司的 MIPS R4000、R5000、R10000 等是超流水线处理机。DEC 公司的 Alpha 处理机是超标量超流水线处理机。

5.3.1.1 基本结构

超标量处理机的典型结构是有多个操作部件,一个或几个比较大的通用寄存器堆,一个或两个高速 Cache。先进的超标量处理机一般都包含有三个处理单元,一个是定点处理单元,通常称为中央处理单元(CPU),它由一个或多个整数处理部件组成;第二个是浮点处理单元(FPU),它由浮点加减法部件和浮点乘除法部件等组成;第三个是图形加速部件,也称为图形处理单元(GPU),这是现代处理机中不可缺少的一个部分。先进的超标量处理机通常都设置有大量的通用寄存器。在有的超标量处理机中,CPU 和 FPU 分别使用两个通用寄存器堆,有的还采用如本书第二章的“RISC 关键技术”一节中介绍的寄存器窗口技术。在多数超标量处理机中都设置有两个一级高速 Cache,一个是指令 Cache,另一个是数据 Cache,这种把指令 Cache 和数据 Cache 分开的结构被称为哈佛(Harvard)结构。每个高速 Cache 的容量一般在几千至几十个千字节;有的超标量处理机,还把二级 Cache 也做在处理机芯片内。

图 5.70 给出由 Motorola 公司生产的一种先进超标量处理机,称为 MC88110。它有 10 个操作部件,其中两个整数部件可以作 32 位的整数运算,也包括地址运算等。整数操作是单周期执行,完成一条整数运算指令使用一条 4 个功能段的流水线,包括取指令 IF、指令译码 ID、执行指令 EX 和写回运算结果 WR,每个时钟周期可以完成两条整数指令。浮点运算是 80 位字长的,包括浮点加、减、乘、除和求浮点平方根等 16 条指令。浮点加法部件和乘法部件都采用 3 级流水线,每个时钟周期可以完成一条乘法指令和一条浮点加法指令;而且,对单精度、双精度及扩展双精度指令的执行速度都一样快。两个专用的图形处理部件可以直接对图形的象素进行处理,它与浮点操作部件一起,提供高性能的三维(3D)图形处理能力,共有 9 条专门的图形处理指令。

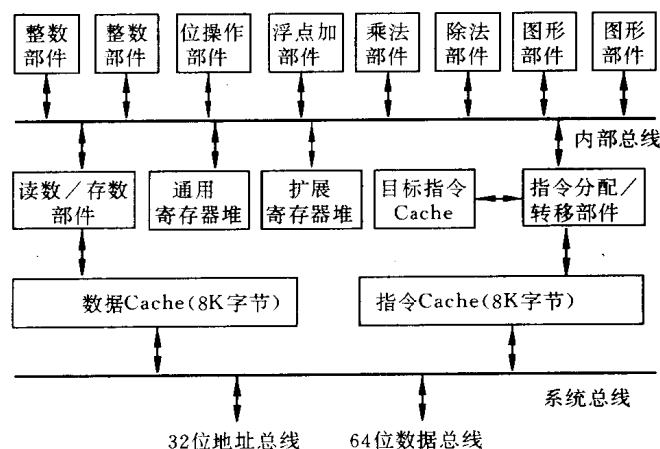


图 5.70 超标量处理机 MC88110 的结构

在 MC88110 超标量处理机内部有两个寄存器堆；其中，整数部件使用通用寄存器堆，它由 32 个 32 位的寄存器组成；浮点部件使用扩展寄存器堆，它由 32 个 80 位的寄存器组成。每个寄存器堆有 8 个端口，分别与 8 条内部总线相连接，可以同时读出 8 个操作数提供给各个操作部件使用。另外，在取数/存数部件中，还有一个缓冲深度为 4 的采用先进先出(FIFO)方式工作的先行读数栈和一个缓冲深度为 3 的也采用先进先出方式工作的后行写数栈。

指令和数据分别存放在两个独立的高速 Cache 中，指令 Cache 和数据 Cache 的容量各为 8K 字节。两个 Cache 都采用两路组相联方式工作，因此，每个时钟周期可以提供两个 64 位的指令和数据。另外，为了减少转移指令对流水线的影响，专门设置有一个转移目标指令 Cache。在遇到条件转移指令时，在指令 Cache 和目标指令 Cache 中分别存放两路分支上的有关指令；并且，指令分配部件在每个时钟周期分别从指令 Cache 和目标指令 Cache 中各取出两条指令来同时进行译码；最后，根据形成的条件码决定把哪一路分支上的指令送到操作部件中去。

5.3.1.2 单发射与多发射

单发射处理机的指令执行时空图如图 5.71(a)所示，它在一个时钟周期内只从存储器中取出一条指令(IF)，并且只对一条指令进行译码(ID)，只执行一条指令(EX)，只写回一个运算结果。

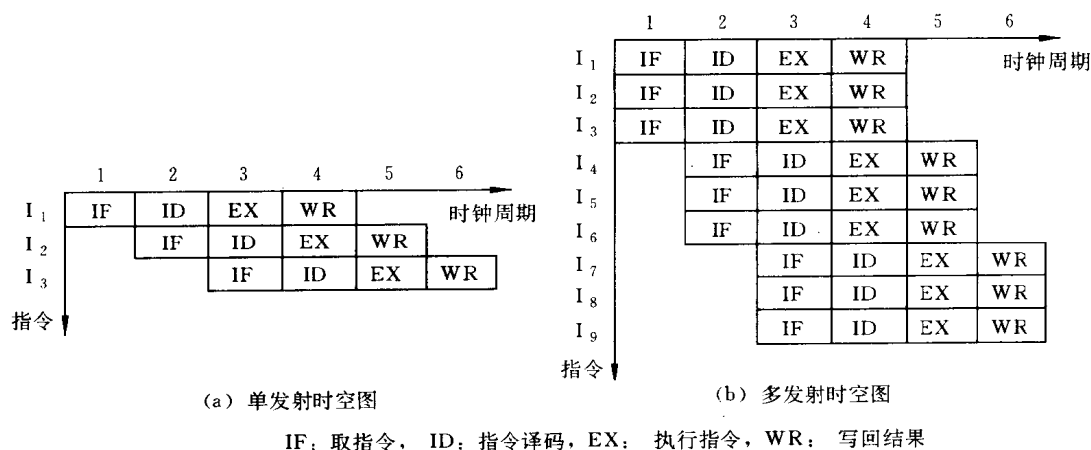


图 5.71 单发射与多发射处理机的指令执行时空图

在单发射处理机中，取指令部件和指令译码部件只各设置一套；而操作部件可以只设置一个多功能操作部件，也可以设置多个独立的操作部件；例如，定点算术逻辑部件 ALU，取数存数部件 LSU、浮点加法部件 FAD、乘除法部件 MDU 等。一个有 4 个操作部件组成的单发射处理机如图 5.72(a)所示。

单发射处理机在指令一级通常采用流水线结构；而在操作部件中，有的机器采用流水线结构，也有的机器不采用流水线结构。

单发射处理机的设计目标是每个时钟周期平均执行一条指令，即它的指令级并行度

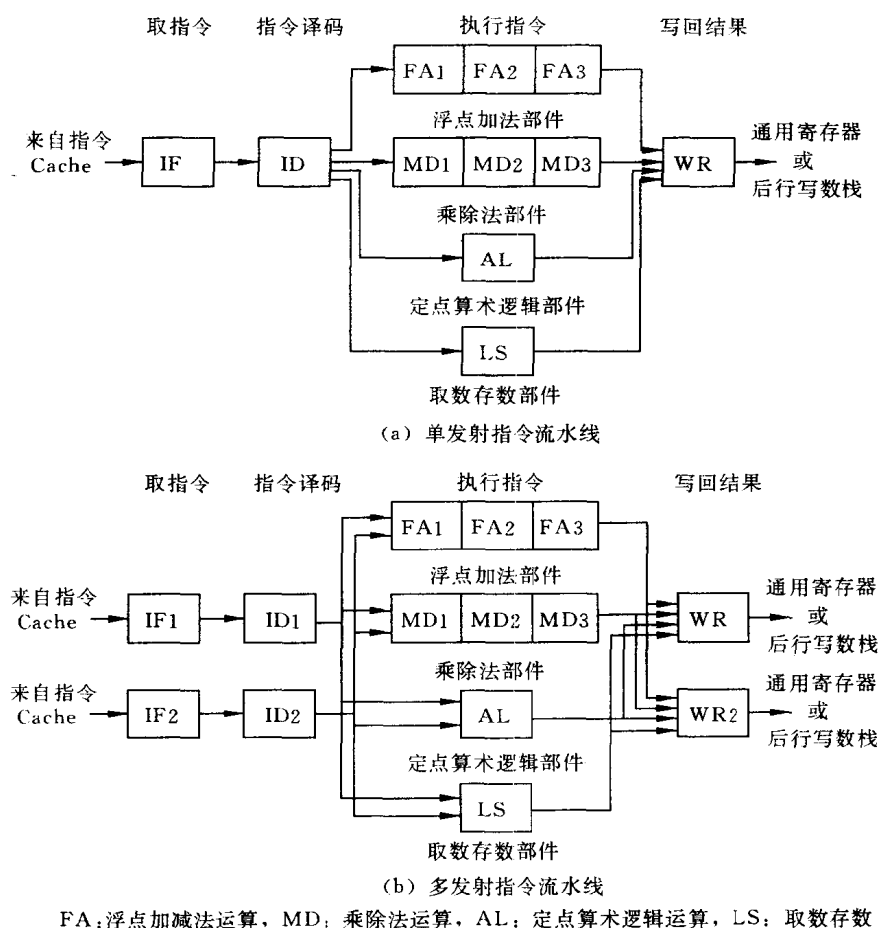


图 5.72 单发射与多发射处理机的指令流水线

ILP 的期望值为 1。如果从表 5.2 中看, 相当于 $m = 1$ 。实际上, 它就是一台有 k 段流水线的普通标量处理机。由于数据相关、条件转移和资源冲突等原因, 实际的 ILP 不可能达到 1。通过优化编译器对指令序列进行重组 (reorganizer), 以及采用软件与硬件相结合的方法处理数据相关、条件转移和资源冲突等, 可以使 ILP 接近于 1; 但是, 单发射处理机的 ILP 不可能大于等于 1。

多发射处理机在一个基本时钟周期内同时从指令 Cache 中读出多条指令, 同时对多条指令进行译码。一个同时发射三条指令的多发射处理机的指令执行时空图如图 5.71 (b) 所示。为了实现在一个时钟周期同时发射多条指令, 通常需要有多个取指令部件, 多个指令译码部件和多个写结果部件。图 5.72 (b) 是一个同时发射两条指令的多发射处理机的指令流水线。两个取指令部件同时从指令 Cache 中取出两条指令, 两个指令译码部件同时对两条指令进行译码, 指令的译码结果分别送往 4 个操作部件执行。

通常, 把一个时钟周期内能够同时发射多条指令的处理机称为超标量处理机。超标量处理机的典型结构如图 5.70 所示, 最基本的要求是必须有两套或两条以上完整的指令执

行部件。图 5.72(b)是典型超标量处理机的指令流水线,为了能够在—个时钟周期内同时发射多条指令,超标量处理机必须有两条或两条以上能够同时工作的指令流水线。高性能超标量处理机通常还有一个先行指令窗口,这个先行指令窗口能够从指令 Cache 中预取多条指令,并且能够对这些指令进行数据相关性分析和功能部件冲突的检测。超标量处理机的基本的要求是在—个时钟周期内能够同时发射两条或两条以上指令。这里应当特别强调同时发射,因为在—个时钟周期内分时发射多条指令的不属于超标量处理机,而是将要在下面介绍的超流水线处理机。

目前,在多数超标量处理机中,每个时钟周期发射两条指令,通常不超过 4 条。由于存在有数据相关和条件转移等问题,采用一般的指令调度技术,理论上的最佳情况是每个时钟周期发射 3 条指令。对大量程序的模拟统计结果也表明,每个时钟周期发射 2 至 4 条指令比较合理。例如,Intel 公司的 i860、i960、Pentium 处理机,Motolora 公司的 MC88110 处理机,IBM 公司的 Power 6000 处理机等每个时钟周期都发射两条指令;美国德州仪器公司(TI)为 SUN 公司生产 SuperSPARC 处理机每个时钟周期发射三条指令。

在有些超标量处理机中,操作部件的个数要多于每个周期发射的指令条数。例如,在许多每个时钟周期发射两条指令的超标量处理机中,通常都有 4 个或 4 个以上独立的操作部件,在有的超标量处理机中有 16 个独立的操作部件。与单发射处理机相同,多发射处理机的操作部件可以采用流水线结构,也可以不采用流水线结构,每个操作部件的时间延迟可以多于—个时钟周期。

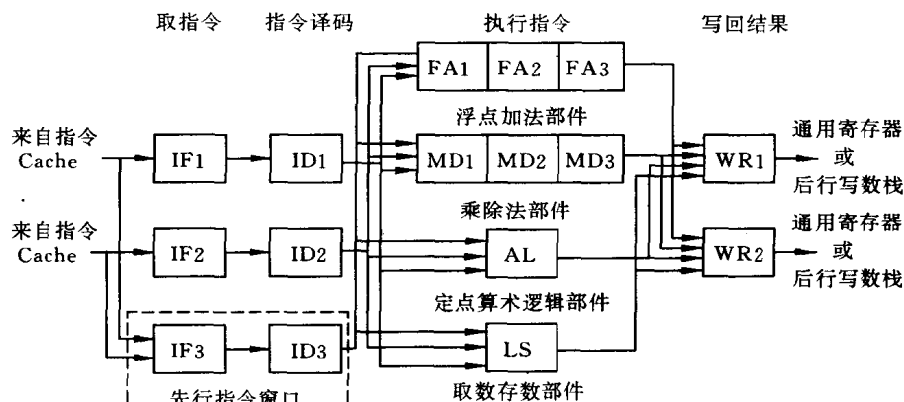
在—个时钟周期内发射多条指令的实现方法可以有多种,使用最多的是在—个存储字中安排两条指令。目前,先进微处理机的字长多数是 64 位,而—条指令的长度—般只有 32 位或更短,因此,在—个存储字中可以安排两条或两条以上指令。即使在 32 位处理机中,虽然主存储器的字长是 32 位,但处理机芯片内的指令 Cache 的字长可以是 64 位。或者更长。有些处理机的指令长度很短,例如,Transputer 系列的 T9000 处理机,最短的堆栈型指令的长度只有 8 位,因此,在—个时钟周期内最多可以发射多达 8 条指令,当然,这需要有优化编译器的配合。

超标量处理机每个时钟周期可以平均执行完成多条指令,因此,它的指令级并行度 ILP 一般都大于 1。如果—台超标量处理机每个时钟周期发射 m 条指令,则它的指令级并行度 ILP 的期望值就为 m 。但是,由于数据相关、条件转移和资源冲突等原因,实际的 ILP 不可能达到 m ,只会小于 m 。通常有: $1 < ILP < m$ 。

在超标量处理机中,不仅需要设置多套取指令部件和指令译码部件,而且要判断指令之间有无功能部件冲突,有无数据相关和由于条件转移引起的控制相关等。另外,还要通过—套交叉开关把几个指令译码器的输出送到多个操作部件中去执行。因此,超标量处理机的控制逻辑是比较复杂的。

当出现数据相关、控制相关或功能部件冲突时,本次没有能够发射出去的指令,必须保存下来,以便在下一个时钟周期再发射。为了提高功能部件的利用率,通常要设置—个先行指令窗口,在这个先行指令窗口中保存由于功能部件冲突、数据相关或控制相关等原因暂时还不能送到操作部件中去执行的指令。—个有先行指令窗口的超标量处理机的典

型结构如图 5.73 所示。



FA:浮点加减法运算, MD: 乘除法运算, AL: 定点算术逻辑运算, LS: 取数存数

图 5.73 有先行指令窗口的多发射流水线处理机结构

先行指令窗口的作用与先行控制技术中的先行指令缓冲栈相似。有了先行指令窗口，就可以从指令 Cache 中读入更多的指令，可以通过硬件来判断哪些指令可以先发射到操作部件中去执行。通过先行指令窗口可以把没有功能部件冲突、没有数据相关和控制相关的指令超越它前面的指令先发射到操作部件中去，从而提高功能部件的利用率。如果再加上编译器的支持，根据先行指令窗口的大小，把没有数据相关、控制相关和功能部件冲突，或者相关和冲突比较少的指令调度到同一个先行指令窗口中，这样，就能够进一步提高超标量处理机的性能。

先行指令窗口的大小对超标量处理机的性能影响很大。窗口太小，调度的效果不好；窗口太大，调度所需要的硬件太复杂。目前，多数超标量处理机的指令窗口大小为 2 至 8 条指令。

在超标量处理机中，有多条指令流水线在同时工作，设置有多条能够独立工作的操作部件，因此，必须解决多流水线的调度问题和操作部件的资源冲突问题。以下两节分别介绍这两个问题。

5.3.1.3 多流水线调度

多条流水线的调度问题非常复杂。已经证明，多流水线调度实际上是一个 NP 完成问题，实现优化调度所需要的代价很大，包括硬件代价和软件代价，通常需要软件（主要是编译器）和硬件的共同结合才能获得比较好的调度效果。本节只介绍一些基本原理和基本的调度方法。

在有多条流水线同时工作时，指令的发射顺序和完成顺序对提高超标量处理机的性能非常重要。如果指令的发射顺序是按照程序中的指令排列顺序进行的，称为顺序发射（in-order issue），否则，称为乱序发射（out-order issue）。同样，如果指令的完成顺序必须按照程序中的指令排列顺序进行，称为顺序完成（in-order completion），否则，称为乱序完

成(out-order completion)。

根据多流水线中指令发射顺序和完成顺序的不同组合,多流水线的调度主要有三种方法,即顺序发射顺序完成,顺序发射乱序完成和乱序发射乱序完成。下面,通过一个具体的程序例子来介绍这三种方法。程序如下:

```

I1: LOAD  R1,  A      ; R1 ← 主存(A)单元
I2: FADD  R2,  R1      ; R2 ← (R2) + (R1)
I3: FMUL  R3,  R4      ; R3 ← (R3) × (R4)
I4: FADD  R4,  R5      ; R4 ← (R4) + (R5)
I5: DEC   R6          ; R6 ← (R6) - 1
I6: FMUL  R6,  R7      ; R6 ← (R6) + (R7)
  
```

在这个由 6 条指令组成的程序中,指令 I₁ 和指令 I₂ 之间有“先写后读”数据相关,指令 I₃ 和指令 I₄ 之间有“先读后写”数据相关,而指令 I₅ 和指令 I₆ 之间除了有“先写后读”数据相关之外,还有“写-写”数据相关。另外,在指令 I₂ 和指令 I₄ 之间,指令 I₃ 和指令 I₆ 之间有功能部件冲突。因此,在这个由 6 条指令组成的短程序中已经包含了所有可能的数据相关和功能部件冲突,这是一个很有代表性的程序。

下面,以这个典型程序的执行过程为例,分别介绍在超标量处理机中所采用的三种不同的指令调度方法。

方法一：顺序发射顺序完成

图 5.74 是采用顺序发射顺序完成的指令调度方法时,上面这个短程序的指令执行时空图。6 条指令按照程序中的指令排列顺序从 I₁、I₂、…、I₆ 分别在流水线 1 和流水线 2 中分三个时钟周期发射。

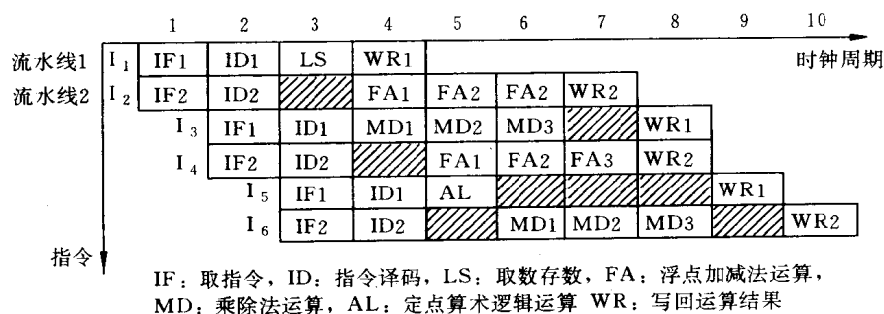


图 5.74 顺序发射顺序完成的指令流水线时空图

由于指令 I₁ 与指令 I₂ 之间有“先写后读”数据相关,指令 I₂ 在流水线 2 中要等待一个时钟周期才能从流水线 1 中通过专用数据通路得到数据;因此,指令 I₂ 在流水线 2 中译码(ID2)完成之后要等待一个时钟周期才能进入浮点加法部件中执行,在图中用阴影部分表示。同样,因为指令 I₅ 与指令 I₆ 之间也有“先写后读”数据相关,因此,指令 I₆ 也要等待一个时钟周期才能进入乘除法部件中执行。

指令 I₄ 在译码完成之后要再等待一个时钟周期才能进入浮点加法部件中执行,这是

因为在指令 I_2 和指令 I_4 都要使用浮点加法器,他们之间有功能部件冲突。

为了维持顺序完成的要求,后发射的指令必须后进入写结果功能段。因此,指令 I_3 在乘除法部件中执行完成之后要延迟一个时钟周期进入写结果功能段。指令 I_2 在定点算术逻辑部件中执行完成之后要延迟三个时钟周期进入写结果功能段。

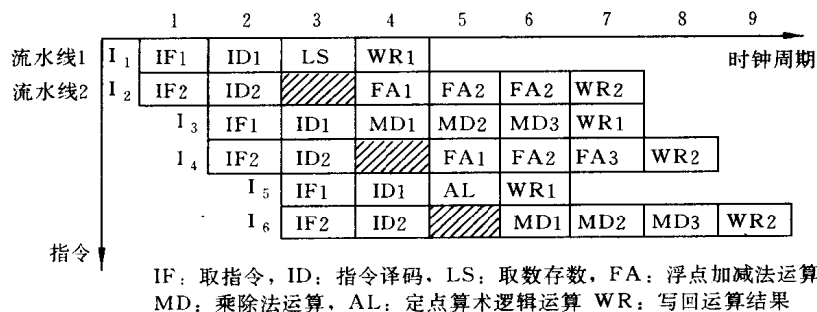
由于指令 I_5 和指令 I_6 之间有“写-写”数据相关,因此,指令 I_6 的写结果功能段要延迟一个时钟周期。

另外,指令 I_3 与指令 I_4 之间虽然有“先读后写”数据相关,由于两条指令在同一个时钟周期中发射,这种数据相关自然得到满足。

从图 5.74 中可以看到,采用顺序发射顺序完成的调度方法,6 条指令共用了 10 个时钟周期才完成。其中,除了流水线的装入和排空部分之外,还有 8 个空闲的时钟周期,在图中用阴影部分表示。在这 8 个空闲的时钟周期中,有 5 个时钟周期实际上是为了维持顺序完成才插入的。

方法二: 顺序发射乱序完成

采用顺序发射乱序完成的流水线执行时序如图 5.75(a)所示。与图 5.74 中的顺序发射顺序完成相比,相同的地方是 6 条指令按照程序中的指令排列顺序分别在流水线 1 和流水线 2 中分三个时钟周期发射,所不同的是,指令在流水线中完成的顺序是混乱的。指令的完成顺序与指令在程序中的排列顺序和在流水线中的发射顺序都无关,6 条指令的完成顺序在图 5.75(b)中给出。



(a) 指令流水线时空图

时钟周期	4	5	6	7	8	9
流水线1	I_1		I_5	I_3		
流水线2				I_2	I_4	I_6

(b) 指令完成次序

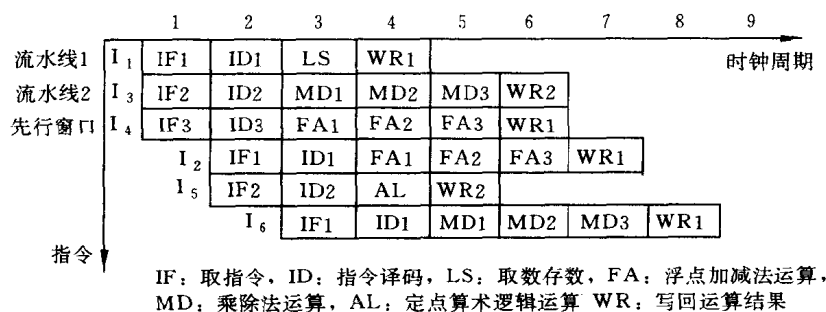
图 5.75 顺序发射乱序完成

从图 5.75(a)中可以看到,只有两个“先写后读”数据相关和一个功能部件冲突,需要流水线空闲等待各 1 个时钟周期。与顺序发射顺序完成调度方法相比,少了 5 个空闲时钟周期。6 条指令总的执行时间为 9 个时钟周期,与顺序发射顺序完成调度方法相比节省了一个时钟周期。因此,采用顺序发射乱序完成的指令调度方法,流水线的总的执行时间和功能部件的利用率都得到了改善。

为了进一步缩短程序的执行时间和提高功能部件的利用率,可以采用图 5.73 所示的先行指令窗口。只要先行指令窗口的大小能够容纳下 6 条指令,就可以通过硬件在先行指令窗口中对这 6 条指令进行数据相关性分析和功能部件冲突的判断,根据分析和判断的结果,对指令进行重新排序,得到一种合理的指令发射顺序。根据这种指令发射顺序,执行程序所需要的总的的时间最短,处理机中功能部件的利用率最高。

方法三: 乱序发射乱序完成

在前面介绍的顺序发射顺序完成和顺序发射乱序完成这两种方法中都没有采用先行指令窗口。如果要采用乱序发射的指令调度方法,就必须使用如图 5.73 中所示的先行指令窗口。图 5.76(a)是采用乱序发射乱序完成指令调度方法时的指令执行时序,图 5.76(b)是所有 6 条指令在流水线中的发射次序。



(a) 指令流水线时空图

时钟周期	1	2	3
流水线1	I ₁	I ₂	I ₆
流水线2	I ₃	I ₅	
先行窗口	I ₄		

(b) 指令流水线发射次序

时钟周期	4	5	6	7	8
流水线1	I ₁		I ₄	I ₂	I ₅
流水线2		I ₅	I ₃		

(c) 指令流水线完成次序

图 5.76 乱序发射乱序完成

由于指令 I₁ 与指令 I₂ 之间有“先写后读”数据相关,通常,指令 I₁ 要早些发射;因此,指令 I₁ 在第一个时钟周期在流水线 1 中发射,而指令 I₂ 在第二个时钟周期也在流水线 1 中发射。指令 I₃ 与指令 I₄ 之间有“先读后写”数据相关,没有功能部件冲突,两条指令可以同时发射,这样,“先读后写”数据相关也就自然消除了。指令 I₃ 在流水线 2 中发射,而指令 I₄ 通过先行指令窗口发射。通常,先行指令窗口除了能够做数据相关性分析和功能部件冲突的检测之外,还应该至少有一套取指令部件和一套指令译码部件。

指令 I₅ 必须在指令 I₆ 之前先发射,这是因为指令 I₅ 与指令 I₆ 之间存在有“先写后读”数据相关。因此,在第二个时钟周期,指令 I₂ 在流水线 1 中发射,而指令 I₅ 在流水线 2 中发射。先行指令窗口不发射指令。

在第三个时钟周期,指令 I₆ 在流水线 1 中发射。

在采用乱序发射时,指令的完成次序必然也是乱序的,6 条指令的完成次序如图 5.76(c)所示。

从图 5.76(a)中可以看出,除了流水线的装入和排空之外,已经没有空闲的时钟周

期,因此,功能部件得到了充分利用。6 条指令总的执行时间缩短为 8 个时钟周期,与顺序发射顺序完成调度方法相比节省了两个时钟周期,与顺序发射乱序完成调度方法相比节省了一个时钟周期。

从程序本身的数据相关性和对功能部件的要求看,采用图 5.73 所示的超标量处理机结构和图 5.76 所示的乱序发射乱序完成的指令调度方法,程序总的执行时间已经是最短的,功能部件的利用率也已经达到最高。

目前,在许多高性能超标量处理机中已经采用了乱序发射乱序完成的指令调度方法。通常设置有一个存储容量为几条指令到十几条指令的比较大的先行指令窗口,一个比较简单的数据相关性分析部件和一个功能部件冲突的检测机构,一般采用计分牌机制来表示数据相关性和功能部件的冲突。另外,通过优化编译器对指令序列进行重组来共同开发程序中指令级并行性。

5.3.1.4 资源冲突

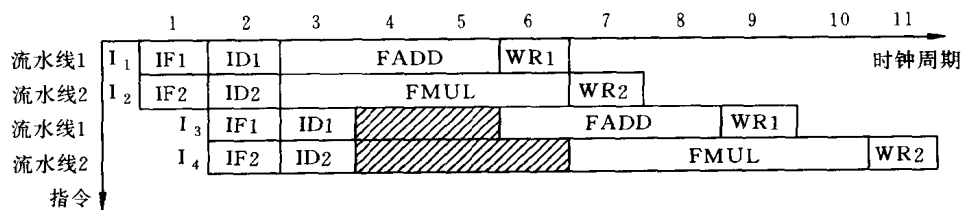
在超标量处理机中,通常设置有多个独立的操作部件。常见的操作部件有定点算术逻辑部件 ALU,浮点加法部件 FADD,乘除法部件 MDU,图形处理部件 GPU,取数存数部件 LSU 等。由于操作部件的数目通常多于每个时钟周期发射的指令条数,因此,这些操作部件可以采用流水线结构,也可以不采用流水线结构。如果采用流水线结构,发生资源冲突(主要是操作部件的冲突)的可能性很小;相反,如果不采用流水线结构,发生资源冲突的可能性就大。

下面是一个由 4 条指令组成的程序:

```

I1: FADD R0, R1      ; R0 ← (R0) + (R1)
I2: FMUL R2, R3      ; R2 ← (R2) × (R3)
I3: FADD R4, R5      ; R4 ← (R4) + (R5)
I4: FMUL R6, R7      ; R6 ← (R6) × (R7)
  
```

如果在一台每个时钟周期发射两条指令的双流水线超标量处理机上执行,有一个独立的浮点加法部件 FADD 和一个独立的浮点乘法部件 FMUL。假设完成一次浮点加法需要 3 个时钟周期,完成一次浮点乘法需要 4 个时钟周期。若浮点加法器和浮点乘法器这两个操作部件都不采用流水线结构,则要发生资源冲突。流水线的时空图如图 5.77 所示。



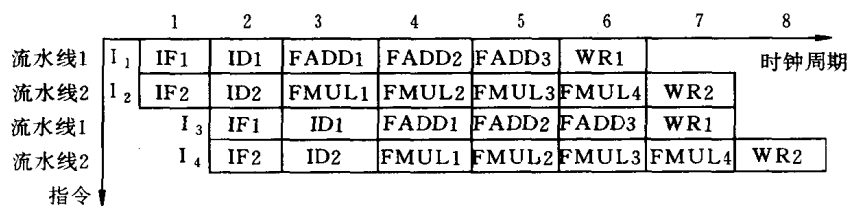
IF: 取指令, ID: 指令译码, FADD: 浮点加法, FMUL: 浮点乘法, WR: 写回结果

图 5.77 双流水线超标量处理机,操作部件不采用流水线的时空图

从图 5.77 中可以看到,由于浮点加法器没有采用流水线结构,指令 I₃ 必须等到指令 I₁ 的浮点加法操作完成之后才能发射到浮点加法部件中去,因此,它在译码完成之后要等

待两个时钟周期。同样,指令 I_4 也必须等待三个时钟周期,到指令 I_2 的浮点乘法操作完成之后才能发射到浮点乘法部件中去。在图 5.77 中,共有 5 个空闲的时钟周期,做完 4 条指令总共用了 11 个时钟周期。

如果浮点加法器和浮点乘法器都采用流水线结构,每一个功能段的时间长度都相等。浮点加法器采用 3 段流水线,浮点乘法器需要 4 段流水线。在一台双流水线超标量处理机上,4 条指令执行的时空图如图 5.78 所示。



IF: 取指令, ID: 指令译码, FADD: 浮点加法, FMUL: 浮点乘法, WR: 写回结果

图 5.78 双流水线超标量处理机,操作部件采用流水线的时空图

从图 5.78 中可以看到,由于浮点加法部件和浮点乘法部件都采用了流水线结构,指令 I_3 和指令 I_4 可以同时发射。指令 I_3 在译码 ID1 完成之后可以直接发射到浮点加法部件中去;同样,指令 I_4 在译码 ID2 完成之后也可以直接发射到浮点乘法部件中去。在图 5.78 中,除了流水线的装入和排空部分之外,没有任何空闲的时钟周期。做完 4 条指令总共用了 8 个时钟周期,与图 5.77 中的浮点加法器和浮点乘法器没有采用流水线的方法相比,少用了 3 个时钟周期。

在超标量处理机中,为了表示资源使用情况和处理资源冲突,要为每一个操作部件设置一个“忙”标志触发器。例如,浮点加法部件 FADD 的“忙”标志触发器为 B_A ,浮点乘法部件 FMUL 的标志触发器为 B_M 。当一条指令译码完成之后,在要送到相应的操作部件中去执行时,必须实现通过资源冲突检测部件检测这个操作部件的“忙”触发器状态。例如,如果 $B_A = 0$,表示浮点加法部件是空闲的,浮点加法操作可以发射到这个部件中去执行。发射完成后,要把 B_A 置成“1”,表示浮点加法部件正在“忙”。当浮点加法操作完成之后,运算结果流出浮点加法部件时,再把 B_A 清为“0”。

比较图 5.77 和图 5.78 可以发现,在一台每个时钟周期发射 m 条指令的超标量处理机中,对于一个延迟时间为 k 个时钟周期的操作部件,如果它不采用流水线结构,则使用同一个操作部件的两条指令的序号应该至少相差 $m \times k$,否则可能发生资源冲突。如果操作部件采用 k 个功能段的流水线结构,则使用同一个操作部件的两条指令的序号只需要相差 m 或 m 以上,就不会发生资源冲突。因为在超标量处理机中,指令流水线的段数 k 一般在 4 至 10 之间,每个时钟周期发射的指令条数 m 在 2 至 4 之间;取中间值, $k = 7, m = 3$;因此,为了不发生资源冲突,如果操作部件不采用流水线结构,两条使用同一个功能部件的指令序号必须相差 21 或 21 以上;如果操作部件采用流水线结构,两条使用同一个功能部件的指令序号只需要相差 3 或 3 以上。从这一分析结果中可以看出,在超标量处理机中,操作部件一般要采用流水线结构,如果由于某种原因,操作部件不能采用流水线结构,则必须设置多个相同种类的操作部件。在许多高性能超标量处理机中,即使操作部件采用

了流水线结构,有些常用的操作部件也设置有多个。例如,图 5.70 中的 MC88110 超标量处理机,整数操作部件就设置有两个。

由于超标量处理机在一个时钟周期内能够发射多条指令,因此,它对指令序列的要求与上一节中介绍的单流水线的标量处理机不同。在单流水线的标量处理机中,只有连续出现相同操作的指令序列时,流水线才能不“断流”,功能部件的效率才能得到充分发挥。而在超标量处理机中正好相反,超标量处理机希望相同操作的指令不要连续出现,否则会发生资源冲突;它要求相同操作的指令能够相对均匀地分布在程序中。超标量处理机的这种要求正好符合一般标量程序的特点。因此,只要超标量处理机中指令流水线的条数和操作部件的个数等设计合理,一般的标量程序能够在超标量处理机上得到高效率的运行,这正是目前超标量处理机能够得到普及的一个重要原因。

5.3.1.5 超标量处理机性能

为了便于比较,把单流水线普通标量处理机的指令级并行度记作 $(1,1)$,超标量处理机的指令级并行度记作 $(m,1)$,超流水线处理机的指令级并行度记作 $(1,n)$,而超标量超流水线处理机的指令级并行度记作 (m,n) 。

在理想情况下, N 条没有资源冲突、没有数据相关和控制相关的指令在单流水线普通标量处理机上的执行时间为:

$$T(1,1) = (k + N - 1)\Delta t \quad (5.50)$$

其中, k 是流水线的级数, Δt 是一个时钟周期的时间长度。

如果把相同的 N 条指令在一台每个时钟周期发射 m 条指令的超标量处理机上执行,所需要的时间为:

$$T(m,1) = \left(k + \frac{N-m}{m} \right) \Delta t \quad (5.51)$$

其中,第一项是第一批 m 条指令同时通过 m 条指令流水线所需要的执行时间,而第二项是执行其余 $N-m$ 条指令所需要的时间,这时,每一个时钟周期有 m 条指令分别通过 m 条指令流水线。

因此,超标量处理机相对于单流水线普通标量处理机的加速比为:

$$S(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{m(k+N-1)}{N+m(k-1)} \quad (5.52)$$

当 $N \rightarrow \infty$ 时,在没有资源冲突,没有数据相关和控制相关的理想情况下,超标量处理机的加速比的最大值为:

$$S(m,1)_{\max} = m \quad (5.53)$$

5.3.2 超流水线处理机

在前面介绍的一般标量流水线处理机中,通常把一条指令的执行过程分解为“取指令”、“译码”、“执行”和“写回结果”4级流水线。如果把其中的每级流水线再细分,例如,再分解为两级延迟时间更短的流水线,则一条指令的执行过程就要经过8级流水线。这样,在一个基本时钟周期内就能够“取指令”两条,“译码”、“执行”和“写回结果”各两条指令。

这种在一个基本时钟周期内能够分时发射多条指令的处理机称为超流水线处理机。在有些资料上把指令流水线的级数为 8 级或超过 8 级的流水线处理机称为超流水线处理机。

超流水线处理机的工作方式与上一节中介绍的超标量处理机不同,超标量处理机是通过重复设置多个“取指令”部件,设置多个“译码”、“执行”和“写回结果”部件,并且让这些功能部件同时工作来提高指令的执行速度,实际上是以增加硬件资源为代价来换取处理机性能的;而超流水线处理机则不同,它只需要增加少量硬件,是通过各部分硬件的充分重叠工作来提高处理机性能的。从流水线的时空图上看,超标量处理机采用的是空间并行性,而超流水线处理机采用的是时间并行性。

5.3.2.1 指令执行时序

一台并行度 ILP 为 n 的超流水线处理机,它在一个时钟周期内能够发射 n 条指令。但这 n 条指令不是同时发射的,每隔 $\frac{1}{n}$ 个时钟周期发射一条指令。因此,实际上超流水线处理机的流水线周期为 $\frac{1}{n}$ 个时钟周期。一台每个时钟周期分时发射 3 条指令的超流水线处理机的指令执行时空图如图 5.79 所示。

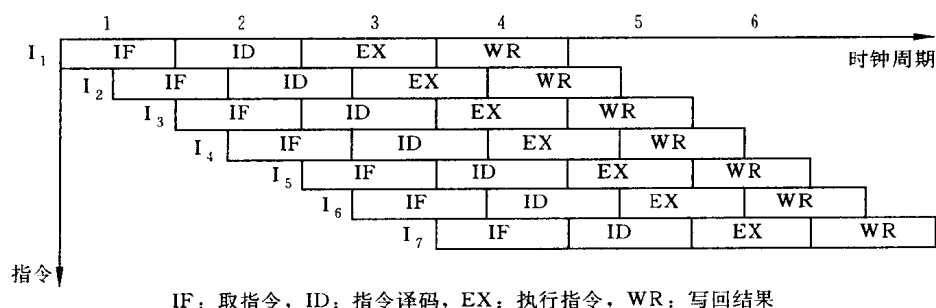


图 5.79 超流水线处理机的指令执行时空图

图 5.79 只是超流水线处理机原理上的指令执行时空图,实际上,功能段还要进一步细分,一个功能段要细分为多个流水级,每一个流水级也都有名称。在分解功能段时要根据实际情况,有些功能段分解的流水级数可多些;例如,图 5.79 中的“译码(ID)”功能段,可以再细分为“译码”流水级、“取第一个操作数”流水级和“取第二个操作数”流水级等;有些功能段分解的流水级数可少些,也的功能段可以不再细分,如“写回结果”功能段一般不再细分。通常,把指令流水线有 8 个或 8 个以上流水级的处理机称为超流水线处理机。

5.3.2.2 典型处理机结构

在早期生产的计算机中,巨型计算机 CRAY-1 和大型计算机 CDC-7600 属于超流水线处理机,其指令级并行度 $n = 3$ 。在目前大量使用的微处理器中,只有 SGI 公司的 MIPS (microprocessor without interlocked piped stages) 系列处理机属于超流水线处理机。MIPS 是除 Intel 公司的 X86 系列微处理器之外,生产量最大的一种微处理器。MIPS 系列的微处理器主要有 R2000、R3000、R4000、R5000 和最近刚投放市场的 R10000 等几种,

其中,R4000 是典型的超流水线处理机。下面以 R4000 为例,说明超流水线处理机的基本结构和工作原理,图 5.80 是 R4000 微处理器的结构框图。

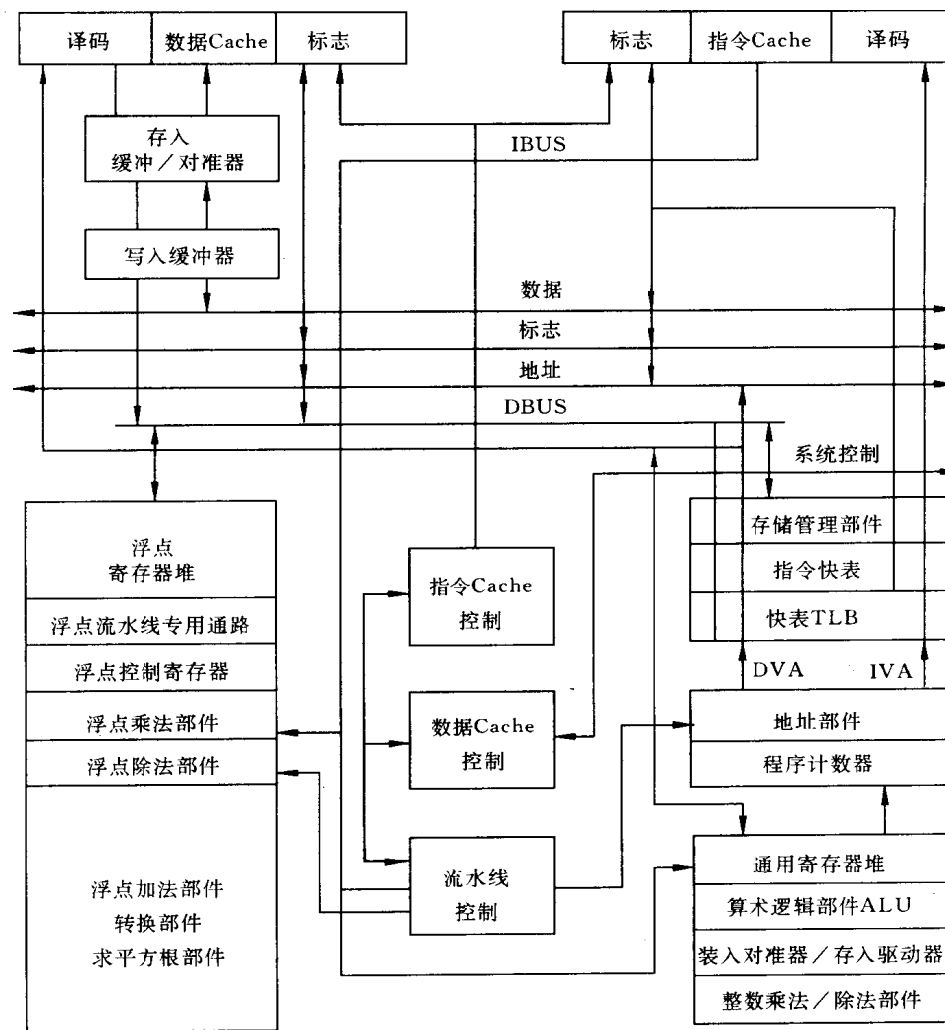


图 5.80 MIPS R4000 超流水线处理机结构

R4000 芯片内有两个 Cache, 指令 Cache 和数据 Cache 的容量各 8K 字节, 每个 Cache 的数据宽度为 64 位。由于每个时钟周期可以访问 Cache 两次, 因此, 在一个时钟周期内可以从指令 Cache 中读出两条指令, 从数据 Cache 中读出或写入两个数据。

整数部件是 R4000 的核心处理部件, 它主要包括 32 个 32 位的通用寄存器, 一个算术逻辑部件 (ALU), 一个专用的乘法/除法部件。整数部件负责取指令, 整数操作的译码和执行, LOAD 与 STORE 操作的执行等。通用寄存器堆用作标量整数操作和地址计算, 寄存器堆有两个输出端口和一个输入端口, 它还设置有专用的数据通路, 可以对每一个寄存器读和写两次。ALU 包括一个整数加法器和一个逻辑部件, 负责执行算术运算操作、地址运算和所有的移位操作。乘法/除法部件能够执行 32 位带符号和不带符号的乘法或

除法操作,它可以与 ALU 并行地执行指令。

浮点部件包括一个浮点通用寄存器堆和一个执行部件。浮点通用寄存器堆由 16 个 64 位的通用寄存器组成,它也可以设置成 32 个 32 位的浮点寄存器。浮点执行部件由浮点乘法部件、浮点除法部件和浮点加法/转换/求平方根部件等三个独立的部件组成,这三个浮点部件可以并行工作。浮点操作主要包括浮点加法、减法、乘法、除法、求平方根、定点与浮点格式的转换、浮点格式之间的转换、浮点数比较等 15 种。浮点控制寄存器用来设置浮点协处理器的状态和控制信息,主要用于诊断软件、异常事故处理、状态保存与恢复、舍入方式的控制等。

R4000 的指令流水线有 8 级,流水线操作如图 5.81 所示。R4000 采用超流水线结构,取指令和访问数据都要跨越两个流水级;实际上,每个时钟周期包含两个流水级,处理器取第一条指令(IF)和取第二条指令(IS)两个流水级都要访问指令 Cache,这两个流水级为一个时钟周期。在寄存器流水级(RF)的开始,指令已经读到了指令寄存器中,因此可以进行译码,并且访问寄存器堆。另外,由于指令 Cache 是采用直接映象方式的,因此,从指令 Cache 中读出的区号要与访问存储器的物理地址进行比较;如果相等,表示指令 Cache 命中。

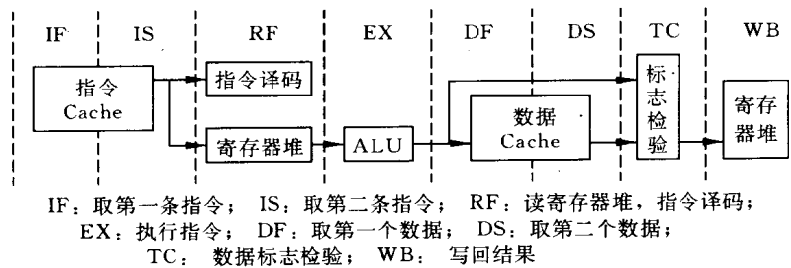


图 5.81 MIPS R4000 处理机的流水线操作

对于非存储器操作指令,如果指令 Cache 命中,那么,指令可以在指令执行(EX)流水级执行,指令的执行结果可以在 EX 流水级的末尾得到。

在正常情况下,MIPS R4000 指令流水线工作时序如图 5.82 所示。一条指令的执行过程经历 8 个流水级(流水级)。由于一个主时钟周期包含有两个流水级周期,因此,也可以认为每 4 个主时钟周期执行完一条指令。

从流水线的输入端看,每一个流水级周期启动一条指令;同样,从流水线的输出端看,每一个流水级周期执行完成一条指令。当流水线被充满时,如图 5.82 中的黑框内所示,有 8 条指令在同时执行。如果把两个流水级周期看作一个时钟周期,则在一个时钟周期内,R4000 处理机分时发射了两条指令,同样,在一个时钟周期内,流水线也执行完成了两条指令。因此,R4000 是一种很典型的超流水线处理机。

在取第一个数据(DF)和取第二个数据(DS)流水级期间,R4000 要访问数据 Cache。首先,存储器管理部件(MMU)在 DF 和 DS 流水级把数据的虚拟地址变换成主存物理地址;然后,在标志检验(TC)流水级从数据 Cache 中读出数据的区号,并把读出的区号与变换成的主存物理地址进行比较;如果比较结果相等,则数据 Cache 命中。对于 STORE 指令,如果命中,只要把数据送到写入缓冲器,由写入缓冲器负责把数据写到数据 Cache 的

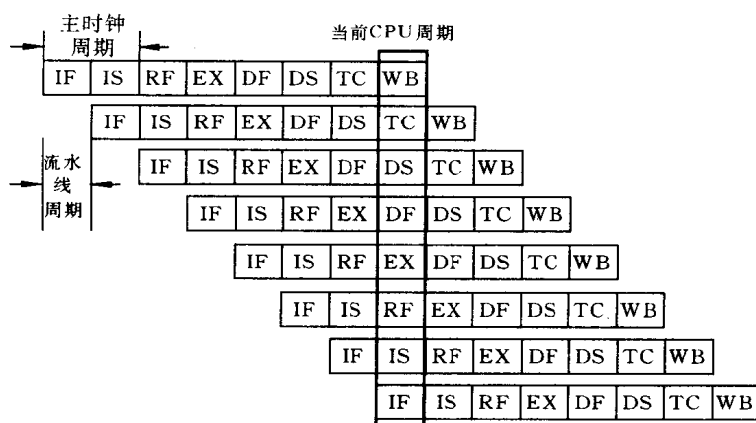


图 5.82 MIPS R4000 正常指令流水线工作时序

指定中去。对于非存储器操作指令，在写回结果(WB)流水级要把指令的最后执行结果写回到通用寄存器堆中。

对于 LOAD 指令，数据要在 DS 流水级的末尾才能准备好。因此，如果在 LOAD 指令之后的两条指令中，任何一条指令要在它的 EX 流水级使用这个数据，则指令流水线要暂停一个时钟周期，如图 5.83 所示。



图 5.83 LOAD 指令引起的流水线暂停

在图 5.83 中，指令 I2 是 LOAD 指令，而指令 I3 要使用指令 I2 读入的数据。在这种情况下，指令 I3 要在 EX 流水级暂停一个时钟周期，同样，指令 I4 和指令 I5 也要暂停一个时钟周期，如图中的阴影部分所示。等到指令 I2 的 DS 流水级完成之后，指令 I2 读入的数据才能够使用；这时，指令流水线又可以继续往前流动。在指令流水线暂停期间，流水级 DF、DS、TC 和 WB 要继续往前流动，而流水级 IF、IS、RF 和 EX 要暂停。如果 LOAD 指令要读入的数据在数据 Cache 中没有命中，则流水线要暂停更长的时间，直到数据从主存储器中读出之后，指令流水线才能继续往前流动。

5.3.2.3 超流水线处理机性能

在一台指令级并行度为 $(1, n)$ 的超流水线处理机上，执行 N 条没有数据相关和控制相关的指令所需要的时间为：

$$T(1, n) = \left(k + \frac{N-1}{n} \right) \Delta t \quad (5.54)$$

其中, k 是指令流水线的功能段数, 或时钟周期数, 而不是流水线级数。在一般超流水线处理机中, 指令流水线的级数实际应为 kn 。(5.54) 式中的头一项是第一条指令通过指令流水线执行完成所需要的时间, 而第二项是执行其余 $N - 1$ 条指令所需要的时间, 这时, 每一个时钟周期有 n 条指令要在指令流水线中执行完成, 也就是每一个流水线周期执行完成一条指令。

单流水线普通标量处理机连续执行 N 条指令所用时间如(5.50)式所示, 因此, 超流水线处理机相对于单流水线普通标量处理机的加速比为:

$$S(1, n) = \frac{T(1, 1)}{T(1, n)} = \frac{n(k + N - 1)}{nk + N - 1} \quad (5.55)$$

当执行的指令条数 $N \rightarrow \infty$ 时, 在没有数据相关和控制相关的理想情况下, 超流水线处理机的加速比的最大值为:

$$S(1, n)_{\max} = n \quad (5.56)$$

5.3.3 超标量超流水线处理机

超标量处理机通过设置多套“取指令”、“译码”、“执行”和“写回结果”等指令执行部件, 能够在一个时钟周期内同时发射多条指令, 同时执行并完成多条指令; 而超流水线处理机则采用把“取指令”、“译码”、“执行”和“写回结果”等功能段进一步细分, 把一个功能段细分为几个流水级, 或者说把一个时钟周期细分为多个流水线周期, 由于每一个流水线周期可以发射一条指令, 因此, 每一个时钟周期就能够发射并执行完成多条指令。

从开发程序的指令级并行性来看, 超标量处理机主要开发空间并行性, 依靠多个操作在重复设置的操作部件上同时执行来提高程序的执行速度。相反, 超流水线处理机则主要开发时间并行性, 在同一个操作部件上重叠多个操作, 通过使用较快时钟周期的深度流水线来加快程序的执行速度。

从超大规模集成电路(VLSI)的实现工艺来看, 超标量处理机能够更好地适应 VLSI 工艺的要求。通常, 超标量处理机要使用更多的晶体管, 而超流水线处理机则需要更快的晶体管及更精确的电路设计。

为了进一步提高处理机的指令级并行度, 可以把超标量技术与超流水线技术结合在一起, 这就是超标量超流水线处理机。下面介绍超标量超流水线处理机的工作原理、处理机结构和主要性能。

5.3.3.1 指令执行时序

超标量超流水线处理机的指令执行时空图如图 5.84 所示。它在一个时钟周期内要发射指令 m 次, 每次发射指令 n 条, 因此, 超标量超流水线处理机每个时钟周期总共要发射指令 mn 条。

在图 5.84 中, 每一个时钟周期分为 3 个流水线周期, 每一个流水线周期发射 3 条指令。从图中可以看出, 每个时钟周期能够发射并执行完成 9 条指令。因此, 在理想情况下, 超标量超流水线处理机执行程序的速度应该是超标量处理机和超流水线处理机执行程序速度的乘积。

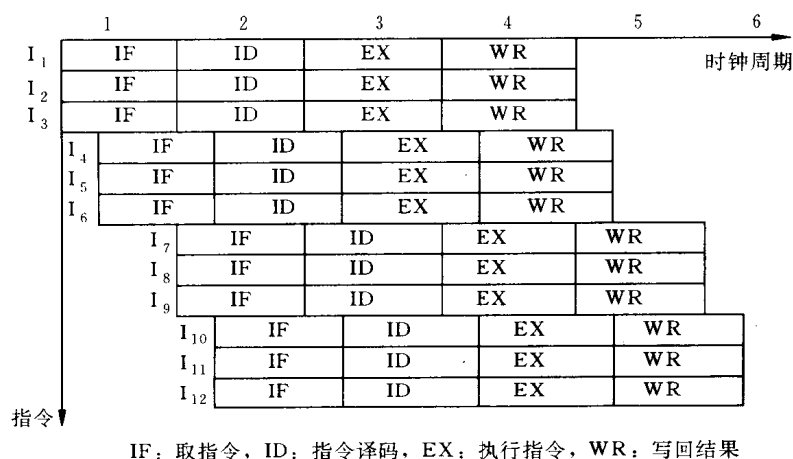


图 5.84 超标量超流水线处理机的指令执行时空图

实际上,图 5.84 只是超标量超流水线处理机原理上的指令执行时空图。在实际的处理机中,IF、ID、EX 和 WR 功能段还要再进一步细分,每个功能段要细分成多个流水级,有些功能段分成的流水级数可能多些,而有些功能段分成的流水级数少些,也的功能段可能不再细分。

5.3.3.2 典型处理机结构

目前,在主要微处理器中只有 DEC 公司的 Alpha 处理机采用了超标量超流水线结构。Alpha 21064 处理机的结构如图 5.85 所示,它主要由四个部件和两个 Cache 组成。四个部件是整数执行部件 EBOX、浮点执行部件 FBOX、地址部件 ABOX 和中央控制部件 IBOX。ABOX 包括地址发生器、存储管理部件、读数缓冲栈和写数缓冲栈。IBOX 负责取指令、指令译码、指令发射、流水线控制、程序计数器 PC 的计算等。

中央控制部件 IBOX 可以同时从指令 Cache 中读入两条指令,同时对读入的两条指令进行译码,并且对这两条指令作资源冲突检测,进行数据相关性和控制相关性分析。如果资源和相关性允许,IBOX 就把两条指令同时发射给 EBOX、ABOX 和 FBOX 三个指令执行部件中的两个。

R4000 处理机采用顺序发射乱序完成的方式控制指令流水线。如果同时从指令 Cache 读入 IBOX 的两条指令中,由于资源冲突或数据相关等原因,第一条指令能够发射,而第二条指令不能发射,则中央控制部件 IBOX 只发射第一条指令。如果第一条指令不能发射,这时,即使第二条指令可以发射,IBOX 也不发射第二条指令,而是让流水线暂停,一直等到第一条指令可以发射时,再启动流水线。

在指令 Cache 中有一个转移历史表,它与中央控制部件 IBOX 中的转移预测逻辑一起实现条件转移的动态预测。在指令 Cache 的每个存储单元中设置有一个“转移历史位”,在执行转移指令时,把转移发生或不发生的情况记录在这个“转移历史位”中,当下次再执行到这条指令时,根据“转移历史位”中记录的信息预测是否发生转移。当一条转移指令第一次被执行时,它还没有转移历史的记录,这时,转移预测是根据指令本身的偏移字段的

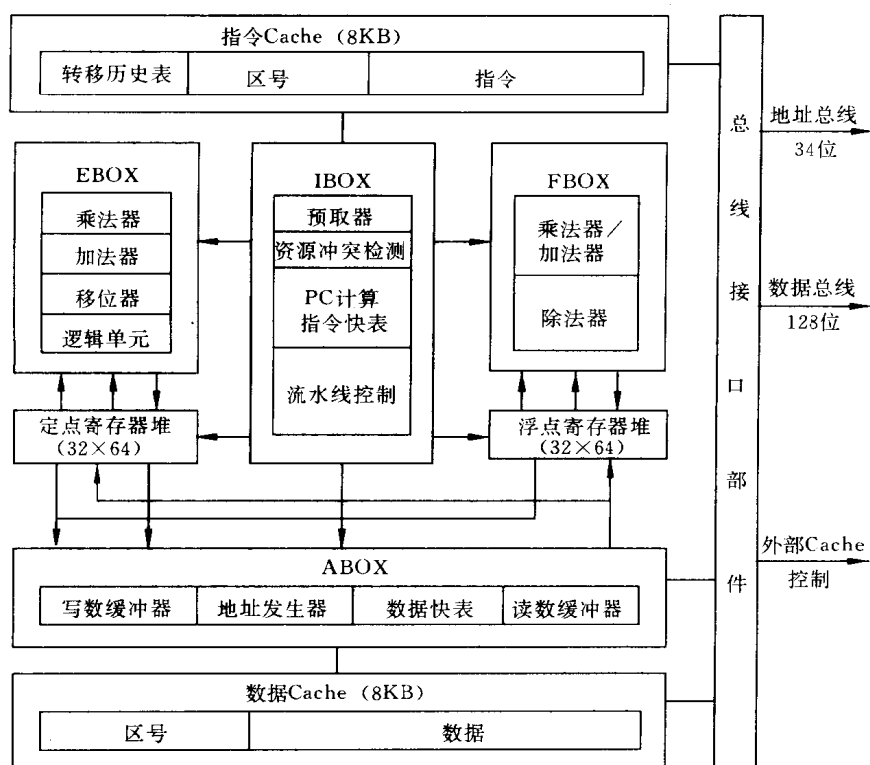


图 5.85 Alpha 21064 处理机结构

符号来决定的。如果偏移字段的符号是负的,则预测转移发生,否则预测转移不发生。当偏移字段的符号是负时,表示要向后转移,通常是转移到一个循环的开始。由于一个循环程序的循环次数一般都大于1,因此,转移发生的概率比较大。

整数执行部件 EBOX 内有一个由 32 个 64 位寄存器组成的定点寄存器堆,这个寄存器堆有四个读出端口和两个写入端口,它可以同时把两个源操作数或结果送到整数操作部件及地址部件 ABOX 中。操作部件的数据宽度为 64 位,包括加法器、桶式移位器、逻辑部件和整数乘法器等。在 EBOX 内还有多条专用数据通路,可以把运算结果直接送到执行部件,而不必先写到寄存器中。

浮点执行部件 FBOX 采用流水线结构,它有两套浮点操作指令,一套是针对 DEC 浮点数格式的,而另一套是针对 IEEE754 浮点数格式的,共有 36 条浮点操作指令。FBOX 内有一个 32×64 位的浮点寄存器堆,这个寄存器堆有三个输出端口和两个输入端口。另外,还有一个用户可以访问的控制寄存器 FPCR。FPCR 包含有舍入控制、陷阱允许和异常事故标志信息等。除了除法指令之外,FBOX 每个流水线周期可以接受一条指令,执行指令的延迟时间是 6 个流水线周期。在 FBOX 内也设置有专用数据通路,当有数据相关时,可以通过专用数据通路把运算结果直接写到执行部件中。

Alpha 21064 芯片内有两个 Cache,一个指令 Cache 和一个数据 Cache。两个 Cache 的容量都是 8K 字节,由于都采用直接映象方式,因此,每个 Cache 中都包含有一个区号字段。另外,由于采用动态转移预测技术,在指令 Cache 中,每个数据块(32 个字节)包含

有一个 8 位的转移历史字段。

Alpha 21064 处理机的指令流水线结构如图 8.82 所示,共有三条指令流水线,每个流水线周期可以发射两条指令。整数操作和地址计算为 7 个流水级,浮点操作为 10 个流水级。流水线都由中央控制部件 IBOX 控制,三条指令流水线的前四个流水级,即取指令 IF、交换双发射指令 SWAP、指令译码 I0 和访问寄存器堆 I1 都在中央控制部件 IBOX 中执行,而流水线的后几个流水级分别在整数执行部件 EBOX、地址部件 ABOX 和浮点执行部件 FBOX 中执行完成。在所有的指令执行部件 EBOX、IBOX、ABOX 和 FBOX 中都设置由专用数据通路,因此,在流水线执行过程中,能够直接把本条指令的操作结果作为下一条指令的操作数使用,而不必先写到寄存器堆中,然后再读出来。

每一条指令流水线实际上是把一条指令的执行过程分成了两部分。一部分是 4 个流水级的静态流水线,在中央控制部件 IBOX 中执行,另一部分是动态流水线。对于整数操作部件 EBOX 和地址部件 ABOX 动态流水线为 3 个流水级,对于浮点操作部件 FBOX 动态流水线为 6 个流水级。

由于资源冲突、数据相关或控制相关等原因,指令在静态流水线中可以停留若干个流水线周期。当指令进入动态流水线之后就必须一直往前流动,不允许停留。因此,每一条指令必须在静态流水线中,即在中央控制部件 IBOX 中完成全部的资源冲突检测,数据相关性和控制相关性分析,在指令从静态流水线发射到动态流水线时,指令整个执行过程所要求的资源、数据相关性和控制相关性都应该是没有问题的。

从图 5.86 中可以看到,Alpha 21064 处理机的三条指令流水线的平均级数为 8 级,而且,每个时钟周期能够发射两条指令。按照一般的定义,每个时钟周期发射多条指令的处理机为超标量处理机,指令流水线的级数为 8 级或多于 8 级的处理机为超流水线处理机,因此,Alpha 21064 处理机应该是超标量超流水线处理机。

(0)	(1)	(2)	(3)	(4)	(5)	(6)
IF	SWAP	I0	I1	A1	A2	WR

IF: 取指令;SWAP: 交换双发射指令,转移预测;I0: 指令译码;I1: 访问通用寄存器堆,发射校验;A1: 计算周期 1,IBOX 计算新的 PC 值;A2: 计算周期 2,查指令快表;WR: 写回整数寄存器堆,指令 Cache 命中/不命中检测。

(a) 整数操作流水线

(0)	(1)	(2)	(3)	(4)	(5)	(6)
IF	SWAP	I0	I1	AC	TB	HM

AC: ABOX 计算有效数据地址;TB: 查数据快表;HM: 写读数据缓冲栈,数据 Cache 命中/不命中检测。

(b) 访问存储器流水线

(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
IF	SWAP	I0	I1	F1	F2	F3	FR	FS	FWR

F1~F5: 浮点计算流水线;FWR: 写回浮点寄存器堆。

(c) 浮点操作流水线

图 5.86 Alpha 21064 处理机的指令流水线

5.3.3.3 超标量超流水线处理机性能

在一台指令级并行度为 (m, n) 的超标量超流水线处理机上, 连续执行 N 条没有资源冲突、没有数据相关和控制相关的指令所需要的时间为:

$$T(m, n) = \left(k + \frac{N - m}{mn} \right) \Delta t \quad (5.57)$$

其中, k 是指令流水线的时钟周期数, 而不是流水线级数。例如, 在 Alpha 21064 超标量超流水线处理机中, $k = 4$ 。 Δt 是一个时钟周期的时间长度。(5.57)式中的第一项是开始 m 条指令通过指令流水线所需要的时间, 第二项是执行其余 $N - m$ 条指令所需要的时间, 这时, 每一个时钟周期平均执行完成 mn 条指令, 也就是每一个流水线周期平均执行完成 n 条指令。

单流水线普通标量处理机连续执行 N 条指令所用时间如(5.50)式所示, 因此, 超标量超流水线处理机相对于单流水线标量处理机的加速比为:

$$S(m, n) = \frac{T(1, 1)}{T(m, n)} = \frac{nm(k + N - 1)}{mnk + N - m} \quad (5.58)$$

当执行的指令条数 $N \rightarrow \infty$ 时, 在理想情况下, 超标量超流水线处理机相对于单流水线普通标量处理机的加速比的最大值为:

$$S(m, n)_{\max} = mn \quad (5.59)$$

图5.87给出超标量处理机、超流水线处理机和超标量超流水线处理机相对于单流水线普通标量处理机的性能曲线。横坐标是三种处理机的指令级并行度。在前面, 曾经用 (m, n) 来表示指令级并行度, 而在图5.87中用 m 与 n 的乘积 mn 表示指令级并行度。纵坐标给出三种处理机的相对性能, 也可以认为是三种处理机相对于单流水线普通标量处理机的实际加速比, 或者认为是这三种处理机所能达到的实际指令级并行度。实际上, 也可以这样来理解图5.87中的曲线, 横坐标表示处理机的设计指令级并行度, 或最大指令级并行度, 而纵坐标表示处理机所能达到的实际指令级并行度。

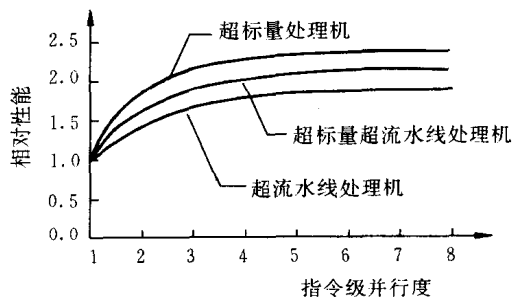


图 5.87 三种指令级并行处理机的相对性能

结合图5.87, 可以得出如下有关结论:

第一, 超标量处理机的相对性能最高, 其次是超标量超流水线处理机, 超流水线处理机的相对性能最低, 主要原因如下:

1. 超标量处理机在每个时钟周期的一开始就同时发射多条指令, 而超流水线处理机则要把一个时钟周期平均分成多个流水线周期, 每个流水线周期发射一条指令。因此, 超流水线处理机的启动延迟比超标量处理机大。

2. 条件转移造成的损失, 超流水线处理机要比超标量处理机大。

3. 在指令执行过程中的每一个功能段, 超标量处理机都重复设置有多个相同的指令

执行部件,而超流水线处理机只是把同一个指令执行部件分解为多个流水级。因此,超标量处理机指令执行部件的冲突要比超流水线处理机小。

第二,当横坐标给出的设计指令级并行度比较低时,处理机实际指令级并行度的提高比较快。但是,当设计指令级并行度进一步增加时,处理机实际指令级并行度提高的速度越来越慢。因此,在实际设计超标量、超流水线、超标量超流水线处理机的指令级并行度时要适当,否则,有可能造成花费了大量的硬件,但实际上处理机所能达到的指令级并行度并不高。目前,一般认为 m 和 n 都不要超过 4。

另外,还有一个图 5.87 中没有表示出来的特性。

第三,一个特定程序由于受到本身的数据相关和控制相关的限制,它的指令级并行度的最大值是确定的。这个最大值主要由程序自身的语义来决定,与这个程序运行在那一种处理机上无关。因此,图 5.87 中的三条曲线,对于某一个特定的程序,最终都要收拢到同一个点上。当然,对于各个不同程序,这个收拢点的位置也是不同的。

一个程序能够达到的实际指令级并行度还与所采用的调度算法有关。目前,国际上已经提出了多种开发指令级并行性的优化调度算法。对于没有条件转移操作,没有输入输出,没有程序调用和程序中断,单入口单出口的基本块程序,实现最优调度并不十分困难。但是,对于一般程序,要充分开发程序中的指令级并行性,实现最优调度非常复杂,已经证明,这是一个 NP 完全问题。另外,实现最优调度所需要的代价很大,包括硬件代价和软件代价,通常需要编译器和硬件的结合才能获得比较好的调度效果。目前,开发程序指令级并行性的许多优化调度算法及编译技术还在进一步研究中。

习 题 五

- 5.1 指令执行过程采用顺序方式、一次重叠方式和流水线方式,它们的主要差别是什么?各有什么优点和缺点?
- 5.2 什么是相关?什么是数据相关?在采用先行控制方式的处理机中,可能有哪几种数据相关?分别是如何解决的?
- 5.3 假设一条指令的执行过程分为“取指令”、“分析”和“执行”三段,每一段的时间分别为 Δt 、 $2\Delta t$ 和 $3\Delta t$ 。在下列各种情况下,分别写出连续执行 n 条指令所需要的时间表达式。
 - (1) 顺序执行方式。
 - (2) 仅“取指令”和“执行”重叠。
 - (3) “取指令”、“分析”和“执行”重叠。
 - (4) 先行控制方式。
- 5.4 在流水线处理机中,可能有哪几种操作数相关?这几种相关分别发生在什么情况下?解决操作数相关的基本方法有哪几种?
- 5.5 在采用先行控制方式的处理机中,有独立的加法操作部件和乘法操作部件各一个,加法操作部件为 4 段流水线,乘法操作部件 6 段流水线,都在第一段从通用寄存器读操作数,在最后一段把运算结果写到通用寄存器中。每段的时间长度都相等,都是

一个时钟周期。每个时钟周期从先行操作栈中发出一条指令。问可能发生哪几种操作数相关?写出发生相关的指令序列,分析相关发生的原因,并给出解决相关的具体办法。

- 5.6 在一台单流水线多操作部件的处理机上执行下面的程序,取指令、指令译码各需要一个时钟周期,MOVE、ADD 和 MUL 操作各需要 2 个、3 个和 4 个时钟周期。每个操作都在第一个时钟周期从通用寄存器中读操作数,在最后一个时钟周期把运算结果写到通用寄存器中。

```
k:      MOVE  R1, R0      ; R1←(R0)
k+1:    MUL   R0, R2, R1 ; R0←(R2)×(R1)
k+2:    ADD   R0, R2, R3 ; R0←(R2)+(R3)
```

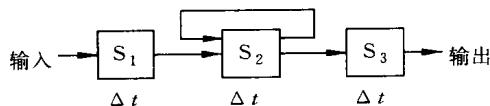
- (1) 就程序本身而言,可能有哪几种数据相关?
- (2) 在程序实际执行过程中,有哪几种数据相关会引起流水线停顿?
- (3) 画出指令执行过程的流水线时空图,并计算执行完这三条指令共使用了多少个时钟周期。

- 5.7 一条线性流水线有 4 个功能段组成,每个功能段的延迟时间都相等,都为 Δt 。开始 5 个 Δt ,每间隔一个 Δt 向流水线输入一个任务,然后停顿 2 个 Δt ,如此重复。求流水线的实际吞吐率、加速比和效率。

- 5.8 用一条 5 个功能段的浮点加法器流水线计算 $F = \sum_{i=1}^{10} A_i$ 。每个功能段的延迟时间均相等,流水线的输出端与输入端之间有直接数据通路,而且设置有足够的缓冲寄存器。要求用尽可能短的时间完成计算,画出流水线时空图,计算流水线的实际吞吐率、加速比和效率。

- 5.9 一条线性静态多功能流水线由 6 个功能段组成,加法操作使用其中的 1、2、3、6 功能段,乘法操作使用其中的 1、4、5、6 功能段,每个功能段的延迟时间均相等。流水线的输出端与输入端之间有直接数据通路,而且设置有足够的缓冲寄存器。用这条流水线计算 $F = \sum_{i=1}^6 (A_i \times B_i)$,画出流水线时空图,并计算流水线的实际吞吐率、加速比和效率。

- 5.10 一条有三个功能段的流水线如下图:



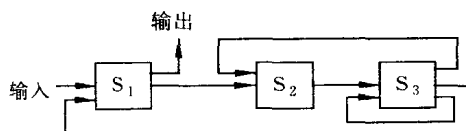
每个功能段的延迟时间均相等,都为 Δt 。其中功能段 S_2 的输出要返回到它自己的输入端循环一次。

- (1) 如果每间隔一个 Δt 向流水线的输入端连续输入新任务,问这条流水线会发生什么情况?
- (2) 求这条流水线能够正常工作的最大吞吐率、加速比和效率。

- (3) 有什么办法能够提高这条流水线的吞吐率,画出新的流水线。
- 5.11 一条有 4 个功能段的非线性流水线,每个功能段的延迟时间都相等,都为 20ns,它的预约表如下:

时间 功能段	1	2	3	4	5	6	7
S_1	×						×
S_2		×				×	
S_3				×			
S_4			×		×		

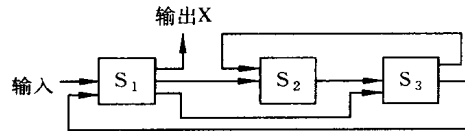
- (1) 写出流水线的禁止向量和初始冲突向量。
 - (2) 画出调度流水线的状态图。
 - (3) 求流水线的最小启动循环和最小平均启动距离。
 - (4) 求平均启动距离最小的恒定循环。
 - (5) 求流水线的最大吞吐率。
 - (6) 按照最小启动循环连续输入 10 个任务,求流水线的实际吞吐率。
 - (7) 画出该流水线各功能段之间的连接图。
- 5.12 一条 3 个功能段的非线性流水线及其预约表如下:



时间 功能段	1	2	3	4	5
S_1	×				×
S_2		×		×	
S_3			×	×	

- (1) 写出流水线的禁止向量和初始冲突向量,并画出调度流水线的状态图。
- (2) 求流水线的最小启动循环和最小平均启动距离。
- (3) 通过插入非计算延迟功能段使该流水线达到最优调度,确定该流水线的最佳启动循环及其最小平均启动距离。
- (4) 画出插入非计算延迟功能段后的流水线连接图及其预约表。
- (5) 画出插入非计算延迟功能段后的流水线状态图。
- (6) 在插入非计算延迟功能段前、后,分别计算流水线的最大吞吐率,并计算最大吞吐率改进的百分比。

- 5.13 对于一条静态非线性流水线,请证明其最小平均启动距离的范围是:
- (1) 最小平均启动距离的下限是预约表中任意一行中“×”的最多个数。
 - (2) 最小平均启动距离的上限是冲突向量中 1 的个数再加上 1。
- 5.14 一条有三个功能段的非线性流水线如下:



- (1) 如果根据流水线各功能部件之间的连接图画流水线的预约表,能画出多少张?流水线的连接图与预约表之间在什么情况下是一一对应的,在什么情况下不是一一对应的?
 - (2) 如果要求所有连接线至少经过一次,试画出该流水线的预约表。
 - (3) 如果规定所有连接线至少经过一次而且只经过一次,画出该流水线的预约表。
- 5.15 一条由 4 个功能段组成的非线性流水线的预约表如下,每个功能段的延迟时间都为 10ns。

时 间 功能段	1	2	3	4	5	6
S ₁	×					×
S ₂		×		×		
S ₃			×			
S ₄				×	×	

- (1) 写出流水线的禁止向量和初始冲突向量。
 - (2) 画出调度流水线的状态图。
 - (3) 求流水线的最小启动循环和最小平均启动距离。
 - (4) 在流水线中插入一个非计算延迟功能段后,求该流水线的最佳启动循环及其最小平均启动距离。
 - (5) 画出插入一个非计算延迟功能段后的流水线预约表(5 行 7 列)。
 - (6) 画出插入一个非计算延迟功能段后的流水线状态变换图。
 - (7) 分别计算在插入一个非计算延迟功能段前、后的最大吞吐率。
 - (8) 如果连续输入 10 个任务,分别计算在插入一个非计算延迟功能段前、后的实际吞吐率。
- 5.16 在指令级并行度 *ILP* 较低时,为什么具有同样 *ILP* 的超标量处理机比超流水线处理机的运行效率要好?
- 5.17 下面一段程序在一台超标量处理机上运行,每个时钟周期发射两条指令。所有指令都要经过“取指令”、“译码”、“执行”和“写结果”四个阶段,其中,“取指令”、“译码”和“写结果”三个阶段的延迟时间都为一个时钟周期。在“执行”阶段,访问存储器部件和逻辑操作部件各延迟一个时钟周期,加法操作部件延迟两个时钟周期,乘法操

作部件延迟 3 个时钟周期,4 种操作部件各设置一个。加法部件和乘法部件都采用流水线结构,每一级流水线的延迟时间都为 1 个时钟周期。每个操作部件的输出都有直接数据通路连接到其它操作部件的输入端。

```

k:      LOAD  R0,  A      ; R0 ← Cache 的(A)单元
k+1:    ADD   R1,  R0     ; R1 ← (R1) + (R0)
k+2:    STORE R1,  B      ; Cache 的 B 单元 ← (R1)
k+3:    ADD   R2,  R3     ; R2 ← (R2) + (R3)
k+4:    MUL   R3,  R4     ; R3 ← (R3) × (R4)
k+5:    OR    R5,  R6     ; R5 ← (R5) ∨ (R6)
k+6:    ADD   R5,  R7     ; R5 ← (R5) + (R7)

```

- (1) 列出程序中可能出现的所有数据相关。
- (2) 采用顺序发射顺序完成调度方法,画出流水线的时空图,并计算执行这个程序所用的时间。
- (3) 采用顺序发射乱序完成调度方法,画出流水线的时空图和各操作的完成时间图,并计算执行这个程序所用的时间。
- (4) 如果再增加一个能够存放 7 条指令的先行指令窗口,采用乱序发射乱序完成调度方法,画出流水线的时空图、各操作的发射时间图和完成时间图,并计算执行这个程序所用的时间。

5.18 在下列不同结构的处理机上运行 8×8 的矩阵乘法 $C = A \times B$, 计算所需要的最短时间。只计算乘法指令和加法指令的执行时间,不计算取操作数、数据传送和程序控制等指令的执行时间。加法部件和乘法部件的延迟时间都是 3 个时钟周期,另外,加法指令和乘法指令还要各经过一个“取指令”和“指令译码”时钟周期,每个时钟周期为 20ns, C 的初始值为“0”。各操作部件的输出端有直接数据通路连接到有关操作部件的输入端,在操作部件的输出端设置有足够容量的缓冲寄存器。

- (1) 处理机内只有一个通用操作部件,采用顺序方式执行指令。
- (2) 单流水线标量处理机,有一条两个功能的静态流水线,流水线每个功能段的延迟时间均为 1 个时钟周期,加法操作和乘法操作各经过 3 个功能段。
- (3) 多操作部件处理机,处理机内有独立的乘法部件和加法部件,两个操作部件可以并行工作。只有一条指令流水线,操作部件不采用流水线结构。
- (4) 单流水线标量处理机,处理机内有两条独立的操作流水线,流水线每个功能段的延迟时间均为 1 个时钟周期。
- (5) 超标量处理机,每个时钟周期同时发射一条乘法指令和一条加法指令,处理机内有两条独立的操作流水线,流水线每个功能段的延迟时间均为 1 个时钟周期。
- (6) 超流水线处理机,把一个时钟周期分为两个流水级,加法部件和乘法部件的延迟时间都为 6 个流水级,每个时钟周期能够分时发射两条指令,即每个流水级能够发射一条指令。
- (7) 超标量超流水线处理机,把一个时钟周期分为两个流水级,加法部件和乘法部件的延迟时间都为 6 个流水级,每个流水级能够同时发射一条乘法指令和一条加法指令。

第六章 向量处理机

本章将讨论向量处理机。向量处理机结构目前已成为解决数值计算问题的一种最重要的高性能结构。它有两个主要的优点——效率高和适用性广。

绝大多数向量处理机都采用流水线结构。当一条流水线不能达到所要求的性能时,设计者往往采用多条流水线。这种处理机不仅能处理单条流水线上的数据,还能并行地处理多条流水线上独立无关的数据。

20 世纪 80 年代出现了许多以流水线运算部件为基础的向量处理机,其中既有价格便宜的与微机相连的协处理机,也有高速的超级计算机。这些超级计算机的计算速度可达 100Mflops 乃至大于 1 000Mflops。

这些向量处理机的性能价格比是很引人注目的,因为与相同价格的串行处理机相比,它们的向量运算吞吐量要高出 1~2 个数量级。但是,这种吞吐量的提高只是对特定结构的问题而言的,也就是说,局限于那些可以转化为向量运算的问题,因为只有向量运算才能充分有效地利用系统的功能部件。

许多超级向量计算机处理通用问题的速度也很快,它相当于一台高性能的串行处理机,但它们处理非向量型问题的吞吐量仅比大多数传统的串行处理机大几倍。如果一台超级向量计算机专门用于处理非向量问题,那么计算成本就会太高,因为其中还包括了向量处理部件的成本,而这些部件进行标量运算时可能是空闲的。

本章的目的在于描述向量处理机的一般结构,并讨论算法和结构如何配合才能高效地处理多类计算机的问题。

6.1 向量处理的基本概念

6.1.1 什么是向量处理

下面我们通过一个简单的例子来说明向量处理与标量处理的差别。先考察一个用 FORTRAN 语言编写的程序:

```
DO 100 I=1, N
  A(I)=B(I)+C(I)
100 B(I)=2 * A(I+1)
```

对上述这个程序循环,在一般的机器上可用以下的指令序列来实现:

```
INITIALIZE I=1
10 READ B(I)
  READ C(I)
  ADD B(I)+C(I)
  STORE A(I)←B(I)+C(I)
```

```

READ A(I+1)
MULTIPLY 2 * A(I+1)
STORE B(I) ← 2 * A(I+1)
INCREMENT I ← I+1
IF I ≤ N GOTO 10
STOP

```

这里常量 2 和数组 A 、 B 和 C 中的每一个元素都称为标量。这一指令序列称为“标量指令序列”，它的执行过程为“标量处理”过程。一般来说，一条标量指令只能处理一个或一对操作数。

上面的程序循环，在向量计算机中通过向量化编译程序得到下面三条向量指令组成的一个向量指令序列：

```

A(1 : N) = B(1 : N) + C(1 : N)
TEMP(1 : N) = A(2 : N+1)
B(1 : N) = 2 * TEMP(1 : N)

```

第一条指令分别取出数组 B 和数组 C 的 N 个元素，并分别相加，然后将 N 个和存入数组 A 。第二条指令将取出的数组 A 的 N 个元素存入暂存区 $TEMP$ 的 N 个单元。第三条指令使暂存区 $TEMP$ 的 N 个元素分别乘 2，并将 N 个乘积存入数组 B 。我们把这 N 个互相独立的数叫做“向量”，对这样一组数的运算叫做“向量处理”。可以看到，一条向量指令可以处理 N 个或 N 对操作数。因此，向量指令的处理效率要比标量指令的处理效率高得多。

6.1.2 向量处理方式

在大型数组的处理中常常包含向量计算，按照数组中各计算相继的次序，我们可以把向量处理方法分为三种类型：

- (1) 横向处理方式：向量计算是按行的方式从左至右横向地进行；
- (2) 纵向处理方式：向量计算是按列的方式自上而下纵向地进行；
- (3) 纵横处理方式：横向处理和纵向处理相结合的方式。

下面以一个用 FORTRAN 语言编写的程序来说明上述三种处理方式：

```

DO 100 i=1,N,1
100  Fi = Ai2 * B + Di * (Ai2 - Ei)

```

1. 横向处理方式

逐个求 F_i 的方式，为此

$$\text{先算: } A_1^2 * B + (A_1^2 - E_1) * D_1 \Rightarrow F_1$$

$$\text{再算: } A_2^2 * B + (A_2^2 - E_2) * D_2 \Rightarrow F_2$$

⋮

$$\text{最后算: } A_n^2 * B + (A_n^2 - E_n) * D_n \Rightarrow F_n$$

一般计算机就是采用这种方式组成循环程序进行处理的。这种处理方式适用于一般的处

理机,即标量处理机,而不适用于向量处理机的并行处理。

2. 纵向处理方式

设 A, D, E, F 是长度为 N 的向量, $A = (a_1, a_2, \dots, a_n)$, $D = (d_1, d_2, \dots, d_n)$, $E = (e_1, e_2, \dots, e_n)$, $F = (f_1, f_2, \dots, f_n)$, 则上述 DO 循环可以写成如下向量运算的形式:

$$F = A^2 * B + D * (A^2 - E)$$

纵向处理方式对整个向量按相同的运算处理完之后,再去执行别的运算。为此

先算	$A^2 \Rightarrow U$
再算	$B * U \Rightarrow V$
再算	$U - E \Rightarrow U$
再算	$U * D \Rightarrow U$
最后算	$U + V \Rightarrow F$

这种处理方式适用于向量处理机。向量长度 N 的大小不受限制,无论 N 有多大,相同的运算都用一条向量指令完成。向量指令的源向量和目的向量都在内存储器中,运算的中间结果需要送回内存储器保存。因而,对存储器的信息流量要求较高。

3. 纵横处理方式

即把长度为 N 的向量分成若干组,每组长度为 n ,组内按纵向方式处理,依次处理各组。若 $N = K \cdot n + r$,其中 r 为余数,也作为一组处理,则共有 $K + 1$ 组,其运算过程为:

先算第 1 组	$A_{1 \sim n}^2 \Rightarrow U_{1 \sim n}$
	$B * U_{1 \sim n} \Rightarrow V_{1 \sim n}$
	...
再算第 2 组	$A_{n+1 \sim 2n}^2 \Rightarrow U_{n+1 \sim 2n}$
	$B * U_{n+1 \sim 2n} \Rightarrow V_{n+1 \sim 2n}$
	...
⋮	⋮
最后算第 $K + 1$ 组	$A_{kn+1 \sim N}^2 \Rightarrow U_{kn+1 \sim N}$
	$B * U_{kn+1 \sim N} \Rightarrow V_{kn+1 \sim N}$
	...
	$U_{kn+1 \sim N} + V_{kn+1 \sim N} \Rightarrow F_{kn+1 \sim N}$

纵横处理方式对向量长度 N 的大小也不加限制,但它是按以 n 为一组进行分组处理的。在每组运算中,用长度为 n 的向量寄存器作为运算寄存器并保留中间结果,从而大大减少了访问存储器的次数。这就可以降低对存储器信息流量的要求,也减少访问存储器发生冲突所引起的等待时间,因而提高了处理速度。

6.2 向量处理机的结构

向量处理机的基本思想是把两个向量的对应分量进行运算,产生一个结果向量。这样,如果 A, B, C 都是向量,各有 N 个元素,则一台向量处理机能够完成如下运算:

$$C = A + B$$

也可以表示成

$$c_i = a_i + b_i \quad 0 \leq i \leq N-1$$

其中 C 用分量形式可表示为 $(c_0, c_1, \dots, c_{N-1})$, A, B 与其类似。

一种采用流水线运算部件实现上述运算的方法如图 6.1 所示。运算器的两条输入数据通路分别传送数据 A 和 B 。存储器每个时钟周期分别提供 A 和 B 的一个元素到相应的输入数据通路上。运算器每个时钟周期产生一个输出值。实际上,数据的输入速率只需和输出速率一样就可以了。如果运算器每 d 个时钟周期输出一个结果,那么输入数据的速率也就只需每 d 个时钟周期为每条数据通路送入一个数据就可以了。

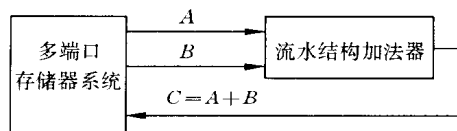


图 6.1 一种能实现两个向量相加的流水结构的加法器

图 6.1 是向量处理机最简单的框图,用它来说明数据在流水线上流动的一般情况。图中流水线运算器是向量计算机的核心部件。

要求向量计算机的存储器系统能提供给运算器连续不断的数据流以及接收来自运算器的连续不断的运算结果,这是设计存储器系统的困难之处。对此,向量处理机在系统结构方面所采用的主要技术都是设法维持连续的数据流,调整操作次序以减少数据流请求。假设取操作数、运算、把结果写回存储器在一个时钟周期内完成,就要求存储系统能在一个时钟周期内读出两个操作数和写回一个运算结果。

一般的随机访问存储器一个时钟周期内最多只能完成一次读操作或写操作。因此图 6.1 所示的存储器系统的带宽至少应 3 倍于一般的存储器系统。这里还忽略了输入/输出操作对存储器带宽的要求,以及取指令对存储器带宽的影响,不过向量结构的一大优点就在于取一次指令可以完成一个很长的向量运算。所以,与传统结构中 20%~50% 的带宽用于取指令的情况相比,向量结构中取指令操作所要求的带宽可以忽略。

系统结构设计者所面临的主要问题是设计出能满足运算器带宽要求的存储器系统。目前市场上出售的向量计算机主要采用两种方法:

(1) 利用几个独立的存储器模块来支持对相互独立的数据的并发访问,从而达到所要求的存储器带宽,即存储器-存储器结构。

(2) 构造一个具有所要求带宽的高速中间存储器,并能实现该高速中间存储器与主存储器之间的快速数据交换,即寄存器-寄存器结构。

在第一种方法中,如果一个存储模块一个时钟周期最多能取一个数据,那么要在一个时钟周期存取 N 个独立数据就需有 N 个独立的存储模块。在第二种方法中,中间存储器的容量较小,所以存取速度比较快,从而获得较高的带宽。但是,由于小容量的存储器中的数据必须由主存装入,尽管其带宽很高,最终大容量的主存仍会成为整个系统的瓶颈。

为了最大限度地利用这种小容量的高速存储器,应尽量多次访问已装入高速存储器的操作数。这样,处理机实际访问主存的请求就会减少,主存的带宽也不必和处理机所要求的最大带宽一样高了。

在后面我们还将看到这种高速存储器的另一个用途是提供主存所没有的访问方式。这样,就可以把矩阵这种数据结构从主存送到中间存储器。矩阵存入中间存储器之后就可

以按行、按列、按对角线或按子阵对其进行快速存取。若矩阵存于主存储器中,就不一定都能按这些方式进行有效的存取。第二种方法在某些机器上又被加以改进,提供不止一级的中间存储器,适当选择各级存储器的容量、成本和性能,使得整个存储器系统的性能价格比较为理想。

6.2.1 存储器-存储器结构

上述的第一种方法如图 6.2 所示。

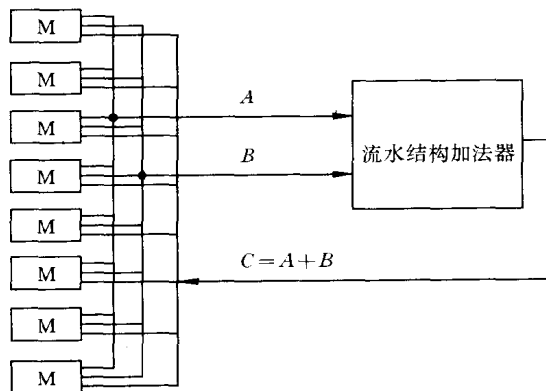


图 6.2 一个具有由 8 个三端口存储器模块组成的存储器系统的向量处理机

主存储器由多个存储器模块构成。图中画的是由 8 个模块构成的存储器系统,它的带宽是单个模块的 8 倍。流水线运算器与主存储器系统之间有三条相互独立的数据通路,各数据通路可以同时工作,不过一个存储器模块在某一时刻只能为一个通路服务。

下面看看这个系统是怎样实现向量运算的。假设一个存储周期占两个处理机周期,那么图 6.2 的存储器系统要满足流水线所需的带宽至少应是单个存储器模块的 6 倍。图 6.3 是计算 $C=A+B$ 最理想的方法。假设向量 A 、 B 、 C 各由八个分量组成。 $A[0]$ 、 $A[1]$ 、 \dots 、 $A[7]$ 分别存放在模块 0、模块 1、 \dots 、模块 7。 $B[0]$ 存放在模块 2, $B[1]$ 存放在模块 3, \dots 、 $B[5]$ 存放在模块 7, $B[6]$ 存放在模块 0, $B[7]$ 存放在模块 1。 $C[0]$ 存放在模块 4, $C[3]$ 存放在模块 7, $C[4]$ 存放在模块 0, \dots , $C[7]$ 存放在模块 3。这样存放便于计算地址。

模块0	$A[0]$		$B[6]$		$C[4]$...
模块1	$A[1]$		$B[7]$		$C[5]$...
模块2	$A[2]$	$B[0]$			$C[6]$...
模块3	$A[3]$	$B[1]$			$C[7]$...
模块4	$A[4]$	$B[2]$			$C[0]$...
模块5	$A[5]$	$B[3]$			$C[1]$...
模块6	$A[6]$	$B[4]$			$C[2]$...
模块7	$A[7]$	$B[5]$			$C[3]$...

图 6.3 向量 A 、 B 、 C 在存储器系统中存放的情况

执行的时序如图 6.4 所示。其中横轴代表时间,纵轴代表存储器模块和流水线部件的工作情况。假设运算流水线分为四段,那么输入数据进入流水线四个时钟周期之后才产生相应的输出值。数据充满之后,流水线就一直处于忙状态。

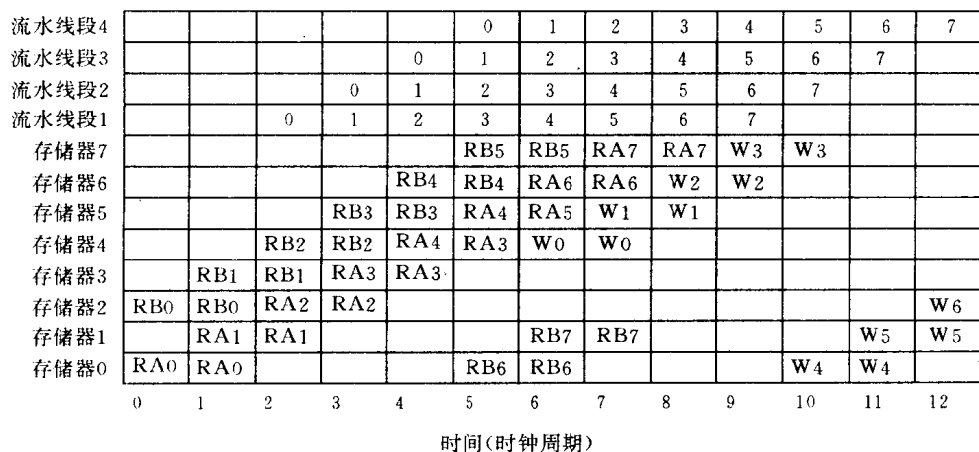


图 6.4 两个向量在流水线方式下分量相加的时序图

图中相应格内用数字标出的段表示正在工作,该数字表示当前周期正在被处理的向量元素的下标。正在进行读操作的存储器模块用“R”表示,后跟一个字符和一个数字,例如符“RA0”表示正在从该模块读出向量 A 的下标为 0 的元素。“W”代表写操作,其后的数字代表写回的向量 C 的相应元素的下标。

在此例中,我们特意将向量按上述方式存放在各个存储模块中,目的是为了防止发生冲突。为了简化讨论,我们还忽略了模块内各元素的寻址问题,而着重讨论使用哪个模块。在时钟周期 0 启动读模块 0 和模块 2 中向量 A、B 的第一个元素,这两个元素在时钟周期 2 被传送到流水线的输入端,在时钟周期 5 结束时得到相应的输出。在时钟周期 1 启动模块 1 和模块 3 中向量 A、B 的第二个元素。在以后每个时钟周期,启动读存在有关模块中向量 A、B 的相应元素。在时钟周期 5 结束时,第一个输出值出现在流水线上。在时钟周期 6,正在读模块 5 和模块 6 中向量 A 的 a_5 和 a_6 元素。在时钟周期 7 开始时模块 5 输出 a_5 ,在时钟周期 8 开始时模块 6 输出 a_6 。在时钟周期 6,正在读模块 1 和模块 2 中向量 B 的 b_5 和 b_6 元素。在时钟周期 6,模块 1,2,3 处于空闲状态。在时钟周期 6,元素 c_0 写入模块 4,在下一个时钟周期元素 c_1 写入模块 5。

在图 6.4 所示的时序中,运算器和存储器的工作衔接得非常好,在整个操作进行过程中没有任何冲突发生。实际情况并非总和上述理想化的例子一样。

如果向量的第一个元素不是存放在我们想存放的存储器模块,那么会发生什么情况呢?例如,当输入向量的位置如图 6.3 所示时,向量加结构决定向量 C 的第一个元素不能存放在模块 0,5,6,7。假如向量 C 在程序的其他地方又作为向量 D 和 E 的和向量,而 D 和 E 在存储器中的存放位置很可能使得向量 C 的第一个元素不能存放在模块 1,2,3,4,这样,我们就会发现 C 所受的限制太多,以至存储器中无位置可以存放 C 来实现无冲突

的存储器操作。

图 6.5 在运算流水线的输入端和输出端增加了缓冲器以便消除争用存储器的现象。例如,假定所有的向量都从模块 0 开始存放,向量运算的时序如图 6.6 所示,向量操作无冲突地进行。图中向量 A 的输入缓冲器延迟两个时钟周期,输出缓冲器延迟四个时钟周期。

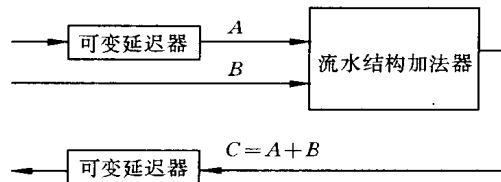


图 6.5 流水结构运算器的输入和输出端增加可变延迟器

从图 6.6 可以看到,先启动读 A 向量,然后再启动读 B 向量。由于在流水结构加法器的 A 向量输入端加了一个缓冲器,使得 A 向量延迟两个时钟周期才能进入流水结构加法器,所以向量 A 和 B 的对应元素同时到达流水结构加法器。在时钟周期 7 结束时第一个结果出现在流水线的输出端,在接着的四个时钟周期模块 0 处于读取 a_8 和 b_8 的状态。向量 A 读出后经过两个时钟周期延迟才进入流水结。向量 A 读出后经过两个时钟周期延迟才进入流水结。

流水线段4								0	1	2	3	4	5
流水线段3							0	1	2	3	4	5	6
流水线段2						0	1	2	3	4	5	6	7
流水线段1					0	1	2	3	4	5	6	7	
存储器7								RA7	RA7	RB7	RB7		
存储器6							RA6	RA6	RB6	RB6			
存储器5						RA5	RA5	RB5	RB5				
存储器4					RA4	RA4	RB4	RB4					RA8
存储器3				RA3	RA3	RB3	RB3					RA8	RA8
存储器2			RA2	RA2	RB2	RB2					RA8	RA8	RB8
存储器1		RA1	RA1	RB1	RB1					RA8	RA8	RB8	RB8
存储器0	RA0	RA0	RB0	RB0					RA8	RA8	RB8	RB8	W0
	0	1	2	3	4	5	6	7	8	9	10	11	12

图 6.6 存储器发生冲突时两个向量相加的时序图

因此,输出缓冲器将每个输出值延迟四个时钟周期之后再送往存储器系统。这样,第一个结果在时钟周期 12 时写入主存。整个向量运算过程比图 6.4 所示的时序延长了 6 个时钟周期。不过,在第一次延迟之后,产生结果和写入存储器的速度是每个时钟周期一个,这与图 6.4 一样。这种在运算器的输入和输出端加缓冲器以消除争用存储器现象的方法,本质上与在流水线内部加缓冲器以消除内部冲突的思想是一致的。

图 6.7 是实现这种思想的方框图。它与 20 世纪 70 年代中期制造的巨型机 CDC STAR 的结构相似。图中一条输入数据通路和结果数据通路分别增加了可变延迟。其延迟时间可根据输入向量和结果向量的第一个元素的位置来设置。这样,在经过初始化阶段

后,流水线将以全速运行。然而,如果向量较短,而延迟时间较长,则会严重影响系统的性能。指令译码器根据向量的起始地址和流水结构运算器执行的具体操作的吞吐率设置延迟的大小。地址生成器形成指令执行过程中要存取的数据的地址。

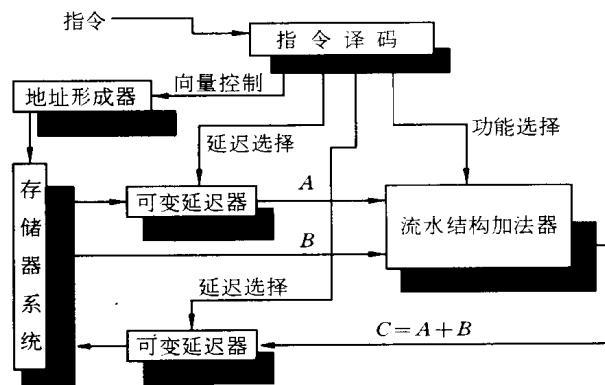


图 6.7 一种与 CDC STAR 相似的系统结构

图 6.7 中的算术运算子系统的功能可以选择。CDC STAR 不能重叠地进行两种或多种向量运算,因此不同的向量操作可以共享共同的运算部件。例如,浮点加和浮点乘就可以使用同一个硬件进行阶码相加、移位以及尾数相加。

CDC STAR 能在一条流水线中进行两个单精度运算或一个双精度运算。如果用单位时间内输出的结果数目来表示输出率,那么单精度的输出率是双精度输出率的两倍。然而,无论是单精度还是双精度,每个单位时间内输出的物理位数是相同的。

图 6.7 中的可变延迟的成本可能较高,而且建立时间也较长。即使成本无关紧要,建立时间却至关重要。

一种能很快地建立某一特定的延迟的方法是使用多抽头的延迟线。数据流由输入端输入,抽头控制器选择某个输出端作为延迟线的输出,如图 6.8 所示。延迟线有 N 个输出端,而网络的实际输出由输出控制决定。D 模块是单位延迟器。延迟量译码器产生的抽头选择信号控制抽头输出。

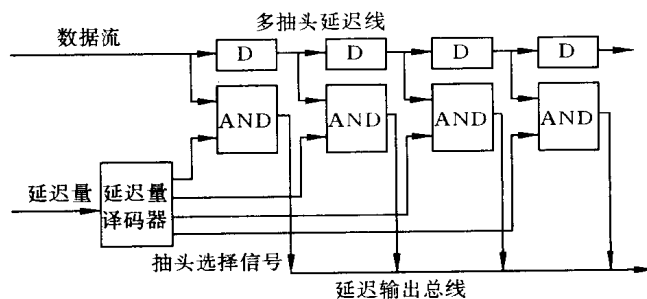


图 6.8 一个由多抽头延迟线构成的可变延迟器

实现可变延迟的另一种方法如图 6.9 所示。这需要一个有 N 个单元的专用存储器。

这个特殊存储器可以同时读某一单元并写另一单元。有两个地址寄存器,一个用于读,另一个用于写。写地址寄存器的初始值是 0,当一个数据写入存储器之后,写地址寄存器的内容增 1。

为了得到 $0 \sim N$ 之间的任意一个延迟值,读地址寄存器的初始值是 $-d$, d 是需要延迟的量。读地址寄存器以操作数到达的速度递增,但在读地址寄存器等于 0 之前没有数据从存储器读出。一旦读地址寄存器等于 0 后,读出的速度和写入的速度就相同了。因此,输出数据流就比输入数据流延迟了 d 个单位时间。

图 6.9 中的存储器有 N 个单元,标号为 $0 \sim N-1$ 。当读地址或写地址的值超过 $N-1$ 时,就复位为 0 继续增值,所以存储器的操作就好像一个循环队列。 N 值的大小只要满足

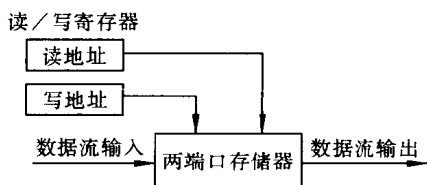


图 6.9 用一个两端口存储器实现的可变延迟器

同步所要求的最大延迟即可。向量操作数可以比 N 长得多,因为延迟存储器在任何时刻都没有必要把整个向量都存起来。

0 延迟的情况很特殊,很容易被检查出来,因为这时读地址和写地址相同。这种情况下输入数据流无需存入缓冲区而直接送往输出端。

图 6.9 所示的可变延迟存储器可将输入流延迟 $0 \sim N$ 个时钟周期。它比多抽头延迟线有

几个优越之处:图 6.9 的可变延迟存储器每个周期最多只有两个地址单元发生状态变化,而多抽头延迟线的每一级都发生变化。

延迟量等于读地址和写地址之差。延迟值为 0 时,输入流通过旁路逻辑(旁路逻辑图中没有画出)直接送往输出端。每改变一次状态,物理量电压或电流都随之变化。每次变化总是需要能量,就会产生热和电噪声。与图 6.8 中延迟器的许多单元在每个时钟周期都发生状态变化的情形相比,图 6.9 的延迟存储器较少的状态变化可能带来较少的瞬态效应和噪声问题。

6.2.2 寄存器-寄存器结构

前面我们已指出使主存有较高带宽的另一种方法是由一级或多级中间存储器形成一个层次结构的存储器系统,其中带宽最高的这一级存储器安排在距处理器最近的位置。当处理器需要向量时,把向量从主存送到速度最快的这一级存储器。中间几级存储器起着把数据送往最快速存储器时或使用后送回主存储器时中间存储的作用,即寄存器-寄存器结构。

Cray 1 系统是美国 Cray 公司于 1976 年提供的产品。它是一台运算速度达亿次/秒以上的巨型机。速度这么高的一个原因是它采用了层次结构的存储器系统。简化的 Cray 1 的框图如图 6.10 所示。主存与流水结构运算器之间有一级或两级的中间存储器。对于向量运算来说,中间存储器是 V 寄存器,它是向量寄存器,它由 8 个 64 个分量的寄存器组成,每个分量为一个 64 位寄存器。向量指令能对向量寄存器的分量进行连续的重复处理。执行向量指令时,流水结构运算器在一个时钟周期内从两个 V 寄存器得到一对操作数,完成某种操作后用了一个时钟周期的时间把结果送入另一个 V 寄存器。主存储器与 V 寄存

器之间的数据传送以成组传送的方式进行。向量流水线是从向量寄存器而不是从主存储器取数据。同样,从流水线输出的结果向量也是送回向量寄存器。

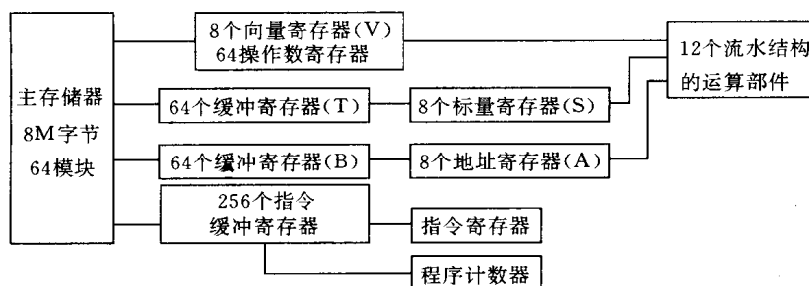


图 6.10 Cray1:一种基于分级存储系统的系统结构

对于标量运算来说,有两级中间存储器。速度很快的一级是 8 个 64 位的 S 寄存器,它是标量寄存器。它们直接与标量运算流水线相连,为标量运算和逻辑运算提供源寄存器和结果寄存器。

另一级速度稍慢一些但仍具很高速的中间存储器是 T 寄存器。它由 64 个标量寄存器组成,每个寄存器字长 64 位。主存储器与 T 寄存器之间以成组传送的方式进行数据传送。由 T 寄存器组成的标量存储器的作用与 Cache 存储器相同,都是为了保存那些在高速的标量存储器中装不下的数据。这些数据有可能暂时不用,但应该保存在离处理器较近的地方以备将来使用,而不应该在两次使用之间又将其送回较远的主存中去。同样,新的数据也可以在被运算器使用之前就预取到中间标量存储器中。

与 Cache 存储器不同的是这些中间存储器不是自动管理的,而是由程序员或编译程序来管理,通过一般的指令将数据装入或移出中间存储器。

这种中间存储器与 Cache 存储器相比的一大优点是速度快。因为流水结构运算器通过寄存器寻址方式访问中间存储器,而访问 Cache 存储器必须查 Cache 表,这需要较长的时间。因此,Cache 存储器的一个周期要完成地址比较操作和通常的读操作。而 Cray1 中的中间存储器则无需花时间去进行类似的 Cache 地址比较操作。

Cray1 系统还有 8 个 24 位的 A 寄存器,它主要用作访问存储器的地址寄存器和变址寄存器,还可用来提供移位的计数值和循环控制值。64 个 24 位的 B 寄存器用作 A 寄存器的中间存储器,它可以存放需要重复访问的数据。例如循环计数值,这时,这些数据就不需要在 A 寄存器或在存储器中保留。主存储器与 B 寄存器之间的数据传送以成组传送的方式进行。这样,B 寄存器组就相当于 A 寄存器组的 Cache 存储器。不过对 B 寄存器的所有操作都是由程序指令直接控制,而不像 Cache 存储器那样是自动控制的。

图 6.10 中还有一个中间存储器,它是指令缓冲器,它由 256 个 16 位寄存器组成,用来存放在指令执行之前就预取出来的指令。由于缓冲器较大,所以主要的程序段可留在其中。内循环指令有可能全部放在指令缓冲器中,这样就可以重复执行而不用再到主存去反复取指令。由于许多应用程序都是紧凑的循环形式,所以取指次数就大大减少。

图 6.10 中流水结构运算器的每个功能部件都有一个高速存储器与之相连。没有一个

功能部件是象图 6.7 所示的处理器直接与主存相连。

设计这种系统结构的主要思想是使操作数离处理器很近,以保证处理器一直处于忙状态。中间存储器能提供给处理器快速存取的数据,而成本又比较低。

中间存储器的性能当然比速度最高的存储器要低。设计这种多级存储器系统时必须对有无中间存储器的性能以及用中间存储器代替高速寄存器能节省多少进行权衡。这里的节省既指成本的降低,也指体积和能量消耗的降低。这在巨型计算机设计中可能是决定性的因素。

中间存储器还能形成新的数据结构,以满足高效处理的要求。流水线一般需要访问向量寄存器中连续的元素,而这些待处理的元素不一定位于存储器的连续单元之中。操作数可先放入中间存储器,再从中间存储器送往向量寄存器。这样,操作数就有可能被重新组织,使下一次要处理的数据被送到向量寄存器的相邻单元。下一节我们将详细地讨论实现这种变换的方法。

本节讨论的两种结构的最突出的特点是存储器和流水线之间密切的配合。第一种结构是依靠主存来保证流水线所需要的操作数。因此主存必须具有至少和运算器所要求带宽一样高的带宽。这就要求主存或者存取速度足够快,或者分为多个独立的存储模块,或者两者都具备,因为运算器要求的最大带宽非常高。

第二种结构是通过容量比主存小得多的中间存储器即寄存器来保证很高的带宽。这样,低速存取的主存就不会妨碍流水结构运算器的连续运行。第二种结构的另一好处是流水结构运算器可以重叠进行,因为高速寄存器的带宽足以满足几个流水结构运算部件的带宽要求。

Cray 1 的流水结构算术运算能重叠地运行,能同时进行三种相互独立的向量运算。一个向量运算所产生的输出可以直接作为下一个向量运算的输入。第一种结构没有提供这种数据通路,所以结果数据流必须首先存入存储器,然后取出再送往运算流水线进行其他的处理。

因为可变延迟器为所有的向量运算所共用,所以缓冲器在下次流水线的延迟建立之前必须排空。流水线在两次运算之间也必须排空,所以重叠操作就没有可能了。Cray 1 能够重叠地流水运算,严格地讲是由于它有中间缓冲器和高速寄存器。

虽然 Cache 存储器是高速计算机的一个很重要的部件,但 Cray 1 却没有 Cache 存储器。Cray 1 设有几个存储器,它们在层次结构的存储系统中的位置与 Cache 存储器类似。没有采用 Cache 存储器的一个原因是向量运算不同于标量运算。

这类机器设计时应考虑非 Cache 结构的中间存储器的编程的成本和难度,也应考虑寄存器寻址方式要比 Cache 访问方式的性能高,必须对两者进行权衡。Cray 1 是为获得高性能而设计的。它的用户都是高级用户,愿意在软件上多花些功夫以获得性能的提高。所以设计时倾向于采用可编程的寄存器组而没采用 Cache 存储器。

另一方面,Cache 存储器对于向量运算可能不如对标量运算那么有效。尽管目前我们还没有太多根据可以证明上述结论,但设计者必须考虑以下这些问题:

- (1) 如果 Cray 1 中的 V 寄存器组采用 Cache 结构,那么命中率会很高吗?
- (2) 向量计算机中的 Cache 容量应该多大?

(3) Cache 的容量是否应足以容纳几个完整的向量?

(4) Cache 的容量是否应小一些,不是容纳几个完整的向量,而是只须容纳多个不同向量的部分分量?

这些问题至今尚无一致的答案,但随着向量技术越来越成熟,在今后几年会得到解决。

Cray 1 的中间寄存器组,包括标量用的 T 寄存器组、地址用的 B 寄存器组和指令缓冲器,都可以用 Cache 来代替。如果这些寄存器组采用 Cache 结构的话,其命中率应该可与一般串行机的命中率相比,但是 Cache 数据的一致性将是一个很困难的问题。

Cray 1 中的有些部件可以修改数据,任何修改必须使存放这些数据的 Cache 能够知道。有些 Cache 一致性协议要求每当一个新数据存入 Cache 时,必须检查所有其他 Cache 中是否已有该数据。这可能引起 Cache 访问冲突而影响性能。

尽管上述方法不是实现互锁地访问 Cache 的唯一途径,但是互锁几乎总是会导致性能的降低和成本的增加。因此 Cache 结构不适合于 Cray 1 这类机器。

看来,将来的设计不必沿着 Cray 1 的方向。器件技术很有可能发生很大变化,使得存储器的集成度、速度和价格都会发生很大的变化。所有这些因素或其中任何一个因素的比较大的变化都有可能产生完全不同的系统结构。随着存储器变得越来越大,越来越快,越来越便宜,有可能出现大容量的中间存储器。

但是,为了使计算机系统的冷却比较容易,要求体积比较小,这就迫使设计者采用小容量的中间存储器,或者根本不采用中间存储器。对巨型机来说,一个比较适当的方法是先保证尽量大的容量和尽量好的性能,然后在不严重影响性能前提下设法减小体积、能量消耗和总的成本。

6.3 向量处理机的存取模式和数据结构

6.3.1 数值算法的存取模式

计算机系统结构必须与负载相适应,这样才能获得高性能。也就是说,一台高速机器必须做它所适于解决的问题,才能达到这种速度。虽然前面几章我们已经提到尽量不要采用太专用的结构,但一台计算机至少应能有效地解决一大类问题。在这一节,我们首先研究某些数值计算问题可能遇到的存取模式以及存取模式对加快算法执行速度的作用。最后我们将讨论怎样设计机器以支持数值计算中经常遇到的存取模式。

假设我们要写一个完成 10^{10} 次乘法的程序,那么有两种极端的情况。一种极端情况是程序员写几百到几千行语句,其中许多语句是调用有关矩阵和向量的库函数。另一种极端情况是程序员为了解决这种非结构化问题而不得不详细地描述 10^{10} 行程序的每一条语句。

很显然,没有人愿意写后一种程序,它需要花费非常长的时间。假设一个人每秒钟能写一个算术操作的程序,那么满负荷工作的话就需 300 年,才能完成整个程序所做的算术表达式的描述,而一台速度为 1 000Mflops 的计算机仅需 100 秒即可完成该程序。

因为许多大型算法可简洁地用矩阵和向量来表示,所以向量和矩阵运算对一台高速

的计算机系统来说至关重要。这一节我们着重讨论向量和矩阵运算问题。

其他表示方式也非常有用,如递归定义的函数。在任何情况下,我们都不能去生成大量非结构化的计算,因为写这种程序要花费大量的时间和精力。

假设我们要求解 $Ax = b$ 的线性方程组,这里 A 为 $N \times N$ 矩阵, x 和 b 为 $N \times 1$ 的列向量。那么采用哪种求解的方法呢?已有的方法和矩阵 A 的性质有关。

若 A 是稠密矩阵,即 A 的全部或几乎全部的元素为非零,则可对矩阵 A 进行一系列行和列的操作, b 在计算过程中也作相应变化,这样可求得方程组的解。

高斯消去法不失为一种求解的有效方法。它将矩阵 A 分解为两个三角矩阵 L 和 U 的乘积。其中 L 为下三角矩阵, U 为上三角矩阵。其特例是,其中 A 为三对角线矩阵, L 和 U 为二对角线矩阵。无论是特殊或一般情况,分解需计算 L 和 U 的元素,这可通过对行和列向量的操作求得。

一旦分解成 L 和 U 后,下一步工作即转化为求三角矩阵方程 $Ly = b$ 和 $Ux = y$ 的解,从而得到满足原方程的解。通过对矩阵 L 和 U 的行进行向量操作,很容易求得三角矩阵方程的解。

若 A 是描述连续性问题的偏微分方程组的矩阵,那么 A 一般是一个极有规则的稀疏矩阵。可采用前一章介绍的循环归约方法求解。我们虽将矩阵 A 视为由行向量或列向量组成,实际情况是描述连续性问题的 A 矩阵的非零元素往往分布在一些对角线上。许多解决这类稀疏矩阵问题的算法经常将矩阵 A 视为由对角线向量组成,这样向量操作的数据是从矩阵 A 的不同对角线上取得。

下面我们研究一个求解线性方程的并行算法。假设一个基本操作即处理矩阵 A 的一行或一列的时间相等。这一假设不是对所有的系统结构都成立,它不仅和计算机的系统结构有关,还和数值计算的程序有关。

算法的核心是 N 次循环的每一循环产生矩阵 L 新的一列和矩阵 U 新的一行。 L 和 U 新产生的数据位于矩阵 A 的相应位置。这些数据在以后的运算中保持不变。 L 和 U 下个新元素产生前,算法将更新 A 矩阵中所有未被更新的元素。矩阵 L 的对角线元素必为 1,所以不必显式存储。矩阵 A 的对角线元素最终被矩阵 U 的对角线元素所重写。

每一次循环,矩阵 A 的对角线上一个元素被重写,我们称此元素为该循环的主元。矩阵 L 的新的一列存放在主元的下面,矩阵 U 的新的行存放在主元的右面。图 6.11 表示一次循环开始时各部分数据的位置。其中 L 和 U 代表矩阵 L 和 U 已计算出的列和行, P 表示主元, L' 和 U' 代表这次循环将要计算出的新值, A 代表矩阵 A 在这次循环中将要被改变的元素。

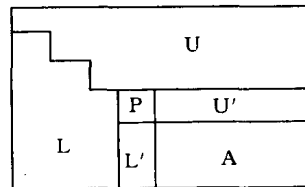


图 6.11 采用高斯消去法在 LU 分解时
某个循环各部分数据的位置

为了保证计算的稳定性,我们应该从 P, L', U' 和 A 中选择绝对值最大的元素作为主元。若此元素不是 P ,我们可通过行列交换,把此元素换到 P 的位置。

若主元是 P 与 L' 组成区域中的最大值,此算法仍能保持稳定。若最大值不在 P 的位

置,则包含该元素的行与主元所在行交换,把该最大值移到 P 的位置。允许行列交换是由于这种交换不会改变原方程组的解。通常情况下,解向量元素需要重新排列,最终产生一个与原问题次序一致的解向量。

程序 6.1 是简化了的高斯消去法的程序。该算法用向量符号表示。符号 $A[i, j]$ 表示矩阵 A 的元素 a_{ij} , $A[1...j-1, j]$ 表示 A 的一个列向量,下标 $1...j-1$ 表示所有 $1 \sim j-1$ 的下标,第二个下标 j 表示第 j 列,所以 $A[1...j-1, j]$ 是由 A 的第 j 列的前 $j-1$ 个元素组成的向量。类似的表示可以代表 A 的一个行向量。

该例有几点需要说明:

- (1) 该算法既要访问行又要访问列。
- (2) 向量操作或在两个向量间进行,或在一个向量与一个标量间进行,结果产生一个新的向量。
- (3) 对一个向量求 MAX 操作所得到的结果是最大元素的下标,而不是最大元素本身。
- (4) 每次循环使被访问向量的长度减 1。

程序 6.1 高斯消去法

FACTOR 是将矩阵 A 分解成上三角矩阵 U 与下三角矩阵 L 乘积的算法。由于所有矩阵的对角线上元素都为 1,故不必显式存储。用 L 重写矩阵 A 的下三角部分,用 U 重写矩阵 A 的对角线和上三角部分。

```
for i := 1 to N do
  begin {找主元的列}
    {求主元行中绝对值最大的元素的下标}
    imax := index of Max(abs(A[i...N, j]));
    {imax 行与 i 行交换,得到 U 的新行}
    Swap(A[i, i...N], A[imax, i...N]);
    {检查奇异性,若是则结束}
    if A[i, j] := 0 then singular matrix;
    {求出 L 的新列并存入 A}
    A[i+1...N, i] := A[i+1...N, i]/A[i, j];
    {更新矩阵 A 的其余部分}
    for k := i+1 to N do
      A[k, i+1...N] := A[k, i+1...N] - A[k, i] * A[i, i+1...N];
  end; {外循环}
```

由此看来,有几点值得提出:

第一点:与我们要求对行列均可访问的假设一致。例中内循环可对行或者对列进行操作,这可由程序员选择。但该算法的其他地方对行和对列操作都需要。所以一个向量机至少能提供对行访问和对列访问的形式,有其他访问形式更好。

第二点:告诉我们一个向量流水线应具有将标量作为操作数的功能,它比两个操作数都是向量的操作更快更有效。

第三点:告诉我们流水结构运算部件应具有最后结果是标量的功能,例如 MAX,

MIN 和 SUM 操作的结果是标量。同时需要注意的是结果标量可能是一个向量的某个重要元素的下标,而不一定是元素本身。

第四点:最复杂。该算法用到的向量每经过一步都要缩短,所以最后一步用的向量的长度为 1。流水线运算和向量操作的开销是确定的,所以我们应处理尽可能长的向量,这样可把开销分摊到各个分量中去。

下面我们介绍解决存取问题的某些技术,对高效的向量机结构作深入的了解。

6.3.2 向量处理机的数据结构

这里将探讨与算法相关数据存取问题。若一个数据结构,如矩阵仅需按行存取,我们可以将相邻元素存放在地址连续的存储器单元。若仅需按列存取,那么我们可以将矩阵按列存储,即将每列的相应元素存放在地址连续的存储器单元。若既要按行存取又要按列存取,则并无明显有效的存放方法能满足这种要求。

对这个问题的进一步说明如图 6.12 所示。主存储器由 8 个相互独立的存储模块组成,一列表示一个模块。图 6.12(a)是一个 8×8 的矩阵存放的形式,同一行的相邻元素存于相邻的存储模块中。这种存放形式有利于流水地访问行元素。

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) 适于访问行向量的存储形式

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)
(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)
(0,7)	(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)

(b) 适于访问列向量的存储形式

图 6.12 一个 8×8 矩阵的两种存储形式

如果完成存储器的一次访问需要几个时钟周期,那么这种存储器在初始延迟后每个时钟周期能提供一个行元素。例如,我们要取出第 0 行向量,先启动读元素(0,0),在该元素被送到主存总线之前,在第二个时钟周期启动读元素(0,1)。在第 i 个时钟周期启动读元素(0, i)。

如果从启动存储器到在输出端口得到数据的时间是 d ,那么对上例来说,元素(0, i)在时钟周期 $i + d$ 结束时才出现在主存总线上。这就是本章开头时介绍过的重叠存取方式。若 d 不大于 8,即不大于例中独立存储模块数,则向量可以任意长。若 d 大于 8,访问长度大于 8 的向量时在某个存储模块会发生冲突。这是因为该模块的上一个访问还没有完成又要求启动新的访问。

从另一角度来说,主存的带宽必须足够高以满足运算器对主存的要求。若延迟时间 d 大于 8,则 8 个存储模块的总带宽不到每个时钟周期输出一个数据,而流水线要求每个时钟周期一个数据。如果延迟时间为 d ,则至少要有 d 个独立的存储器模块才能使总带宽达到每个时钟周期输出一个数据的要求。若一条指令需要三个操作数,其中两个是源操作数,一个是目的操作数,则主存的总带宽至少要达到每个时钟周期输出三个数据,存储器模块的数目至少要 $3d$ 个才能满足每个时钟输出一个结果的流水线速度。

图 6.12(a)告诉我们,仅主存带宽满足要求还不够。例如欲存取矩阵的某列元素,比如第 0 列,会发生什么情况呢?第 0 列的元素存放在同一个存储模块中,不管整个系统有多少个存储模块,存取某列元素将受这个存储模块的带宽的限制。在这种情况下,每 d 个单位时间才能取得一个数据。

如果把矩阵按如图 6.12(b)所示的形式存放,那么同一列的元素存放在不同的存储模块中,访问列向量的速度就大大提高,但访问行向量的速度降低了。上一节介绍的高斯消去算法既要访问行向量又要访问列向量,所以图 6.12(a)和 6.12(b)所示的存储形式均不能满足要求。解决问题的方法之一是修改算法使之仅访问行向量或列向量。这对高斯消去法也许可行,但对有些算法就不一定可行。

另一方法是改变数据的存储结构。图 6.13 是一个 8×8 矩阵的另一种存储形式:相邻行之间错开一位。第 0 行从第 0 个模块开始存放,第 1 行从第 1 个模块开始存放,……,每行相对于上一行右移了一列。

按这种存储形式,一个数据在系统空间中的地址为 $8X$ (本地地址) + 存储模块号,其中存储模块号为 $0 \sim 7$ 。同一行相邻元素的地址是连续的,而同一列相邻元素的地址相距 9。由于同一列元素存放在不同的存储模块中,所以与访问行元素一样也可以按流水方式存取。

虽然矩阵是 8×8 ,我们把它当作 8×9 (8 行 9 列)的矩阵来存储,存放第 9 列元素的单元浪费了。这一附加的列使得列元素与行元素一样被分布在所有的存储模块中。

如果类似于图 6.7 和图 6.10 所示的向量计算机采用上述这种存储结构,那么向量操作数必须由下列四个量来确定:

- (1) 起始地址;
- (2) 元素个数;
- (3) 精度(每个元素的位数);
- (4) 步距(两个相邻元素间的偏移量)。

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(1,7)		(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(2,6)	(2,7)		(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(3,5)	(3,6)	(3,7)		(4,0)	(4,1)	(4,2)	(4,3)
(4,4)	(4,5)	(4,6)	(4,7)		(5,0)	(5,1)	(5,2)
(5,3)	(5,4)	(5,5)	(5,6)	(5,7)		(6,0)	(6,1)
(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)		(7,0)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	

图 6.13 一种允许按行访问和按列访问的数据结构
(按行访问的步距为 1,按列访问的步距为 9)

向量的步距表示向量存取过程中,从一个元素到下一个元素的地址增量。在图 6.13 中,行存取的步距为 1,列存取的步距为 9。通常情况下,如果步距与存储模块数 M 互质,则 M 次按该步长的连续访问是访问 M 个不同的存储模块。更一般的情况下,若存储模块数 M 和步距 S 为任意值,则 M 次按步距 S 的连续访问是访问 $M/\text{GCD}(S, M)$ 个不同的存储模块。GCD 是求最大公因子函数,当两个变量互质时,最大公因子等于 1。

因为 M 通常为 2 的幂,所以 M 次步距为奇数的连续访问是访问 M 个不同的存储模块。如果 $11 \times 11, 13 \times 13$ 矩阵如图 5.13 那样存放,我们很容易证明按行访问和按列访问像 9×9 矩阵那样容易。若矩阵有偶数列,则访问列向量时会发生冲突,这是因为偶数与 M 不互质。例如,一个 12×12 矩阵当 d 大于 2 时将发生冲突,因为列元素 1, 3, 5, ..., 存放在同一存储模块中。同样的原因,访问 $8 \times 8, 24 \times 24$ 矩阵列向量的效率很低。

幸运的是偶数加 1 就变成奇数。因此,我们总可以在一个数据结构上加上一列无用列,从而变成适合于流水地访问行和列的存储结构。

对矩阵的访问除按行和按列外,还有按主对角线访问和按副对角线访问等形式。访问矩阵主对角线元素的步距比访问列元素的步距大 1。若存储模块数 M 是 2 的幂,则访问列元素和访问主对角线元素不可能同时都有效,因为两个步距必有一个与 M 不互质。

有人曾提出一个令人吃惊的方案——存储器模块数目不是 2 的幂。例如,存储器模块数目是质数 P ,则所有小于 P 的步距都与 P 互质。因此,我们可以把矩阵存放在这种按行、列、对角线访问都同样有效的存储器里。BSP 计算机采用了这种方法,它的结构如图 6.14 所示。

BSP 计算机有 17 个存储器模块(不是 16 个),这些模块构成共享的多体并行存储系统,它通过高速互连网络与 16 台处理器相连接。BSP 把连续几条向量指令在时间上重叠起来执行,构成一条五级的数据流水线。五级的功能作用依次是:由 17 个存储器模块并行读出 16 个操作数;经输入对准网络将 16 个操作数重新排列成 16 台处理器所需要的次序;将排列后的 16 个操作数送 16 台处理器完成操作;所得 16 个结果经输出对准网络重

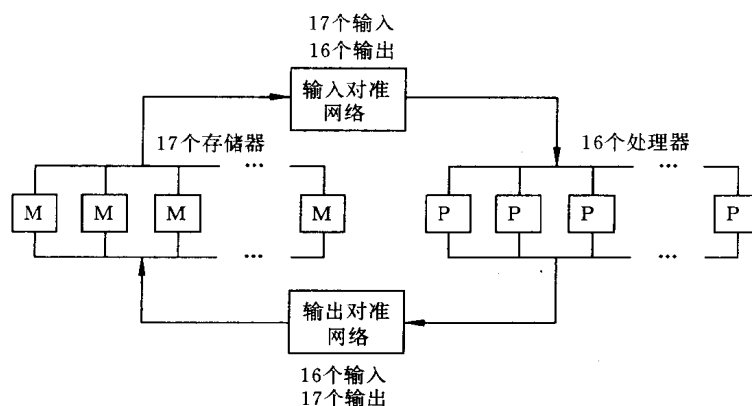


图 6.14 BSP 计算机五级数据流水结构示意图

新排列成 17 个存储器模块所需的次序；最后写入存储器。对准网络的作用是使存储器中为保证无冲突访问而错开存放的操作数次序与并行处理器的正常次序保持协调配合。

图 6.15 表示一个向量从 5 个存储器模块读出经过对准网络后送往 4 台处理器的过程。图 6.15(a)是从一个 4×4 矩阵中读出一个列向量的过程。图 6.15(b)是从一个 4×4 矩阵中读出主对角线元素的过程。

按图 6.15 形式存放一个 4×4 矩阵，访问行元素的步距为 1，访问列元素的步距为 6，访问对角线元素的步距为 7。因为 1, 6, 7 与 5 互质，所以上述几种访问形式均不会出现存储器冲突。

BSP 这种 17 个存储体的结构解决了存取中的一些问题，但又产生了其他新的问题。这种结构的寻址比 M 为 2 的幂的存储系统的寻址复杂得多。另外，17 个存储体的存储系统要求被访问的矩阵元素必须存放在远离处理器的主存中，而不是存放在离处理器很近的缓冲区内。所以仅仅为了把数据的格式对准，来回于主存储器的信息流量已经非常大了。

下面把 17 个存储体的结构与图 6.10 所示的 Cray 结构作一比较。Cray 结构是高速缓冲存储器(向量寄存器)驱动算术部件，而 17 个存储体的结构是主存驱动算术部件，其中两个对准网络更延长了存储周期。Cray 系统能将数据在使用之前就取到高速缓冲存储器。

当数据已在高速缓冲存储器，在写回主存之前可被多次运算。可以想象有一个足够大的高速缓冲存储器将矩阵的很大一部分元素装入，可按多种访问形式中的任何一种进行访问。目前这种高速缓冲存储器的成本非常高。一个非常简单的替代方法是：矩阵元素重定格式是在主存与高速缓冲存储器之间传递信息过程中完成。例如，一个 8×8 矩阵按行存放在由 8 个存储体构成的存储器系统中。若算法的某一步要按列进行访问，可以先将行向量装入高速缓冲存储器，通过重定格式，将 8×8 的形式变成 8×9 的形式，再将 9 个元素的行向量写回主存。经过重定格式的向量可以写回主存的另一区域，以避免将原矩阵冲掉。由于行操作是流水线方式进行，所以读一行 8 个元素比读一个元素稍费一点时间。矩阵写回主存后，就可以对它以步距 9 按列访问了。重定格式的时间近似地等于传递 2~4 个向量所需的时间。按列访问经过重定格式的矩阵要比按列访问原矩阵快 d 倍，其中 d 是存储器存取周期。

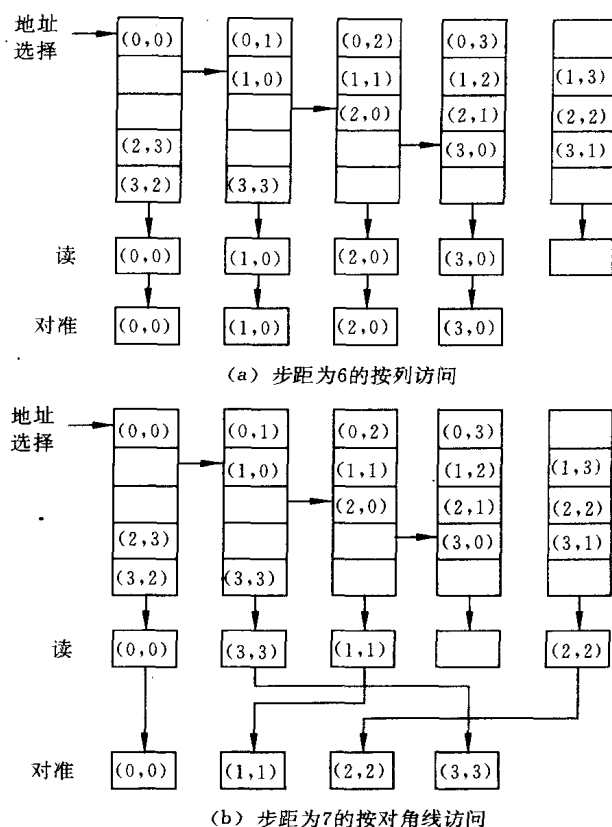


图 6.15 一种按行、按列和按对角线存取效率都较高的数据结构

用上述方法进行重定格式比用图 6.15 所示的对准网络进行重定格式的开销要小,因为上述重定格式仅在需要时才进行,而对准网络使每次向量存取操作都延迟了。

看来具有高速缓冲存储器的系统结构优于那种主存与处理器直接相连的系统结构。尽管这一看法很大程度上和现有的技术水平有关,但现在的趋势是利用高速缓冲存储器来提高向量处理机的速度,而不是把所需的操作数存放在远离处理器的地方。高速缓冲存储器设计中的主要问题是构造一个既足够大(即能存储相当数量的数据)、又足够快(即能按运算部件的时钟周期存取)的存储器。当高速缓冲存储器的容量减小时,数据在放回主存之前被用来进行计算的次数就减少,所以一个容量很小的高速缓冲存储器对减少访问主存的次数的作用不大。

6.4 提高向量处理机性能的方法

6.4.1 向量处理机系统结构的设计目标

研制高性能向量处理机要解决以下几个主要问题:

1. 较好地维持向量/标量性能平衡

向量处理机除具有处理向量功能外还需要具有处理标量的功能,但处理这两类计算

必须平衡。

向量平衡点(vector balance point)定义为为了使向量硬件设备和标量硬件设备的利用率相等,一个程序中向量代码所占的百分比。换句话说,我们希望花在向量硬件和标量硬件上的时间相等。这样,资源就不会空闲。

如果系统在向量模式下能够达到 9Mflops、在标量模式下能够达到 1Mflops 的运算速度,假设代码的 90%是向量运算,10%是标量运算,这样花在两种模式上的计算时间相等,那么向量平衡点为 0.9。

一个系统执行向量运算和标量运算的时间相等并不一定算是最好。但是应该保持足够高的向量平衡点,以便与用户程序的向量化程度相匹配。

每台处理机重复设置流水线功能部件可以提高向量运算性能。另一种方法是向量部件采用超流水线技术,其时钟频率是标量流水线操作的两倍或三倍。

要真正获得所期望的性能则需要较长的向量。可以设想到 2000 年能做出一种同时运行多个功能部件、峰值速度达到 8Gflops 的处理机。

图 6.16(a)和 6.16(b)是在 Cray 超级计算机和日本超级计算机上运行 livermore Fortran 循环时,单处理机的向量运算性能和标量运算性能。所有超级计算机的标量性能沿着图中的虚线上升。

日本超级计算机的向量性能显然优于 Cray 公司的机器。其原因之一是采用了高时钟频率,其他原因还有用了更好的编译器及优化支持。

表 6.1 对 7 种超级计算机的向量和标量性能做了比较。大多数超级计算机的向量平衡点在 90%或更高。向量/标量的比例越高,对目标代码向量化比例的依赖也越大。

表 6.1 各种超级计算机向量和标量的性能

机器型号	Cray 1S	Cray 2S	Cray X-MP	Cray Y-MP	Hitachi S820	NEC SX2	Fujitsu VP4000
向量性能(Mflops)	85.0	151.5	143.3	201.6	737.3	424.2	207.1
标量性能(Mflops)	9.8	11.2	13.1	17.0	17.8	9.5	6.6
向量平衡点	0.90	0.93	0.92	0.92	0.98	0.98	0.97

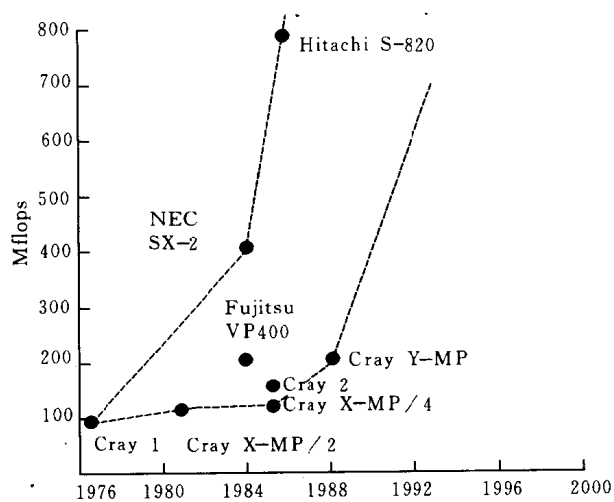
IBM 向量机的设计思想与上述方法不同,它维持较低的向量/标量比例,定在 3~5 的范围之间。这是综合考虑了通用应用问题对标量和向量处理要求的结果。

2. 可扩展性随处理机数目的增加而提高

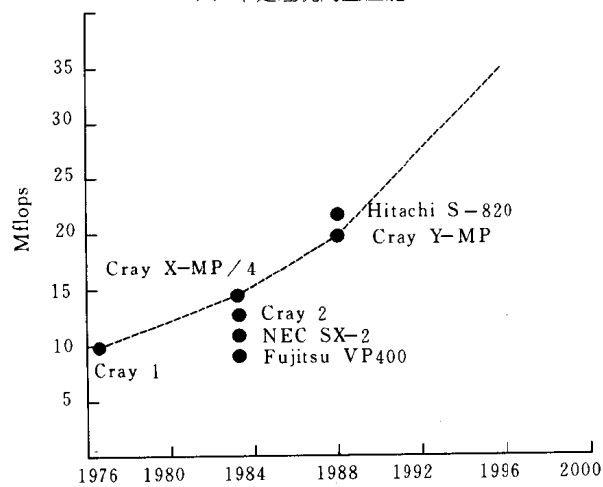
可扩展性的最简单的定义是在确定的应用背景下,向量处理机系统性能要随处理机数目的增加而线性地提高。可扩展性的三个目标是:

- **规模可扩展性** 一台规模可扩展的计算机应设计成其资源部件个数从小到大可扩展,这里所说的部件包括处理单元、存储器等。我们不能单为了达到规模可扩展而不考虑价格、效率。客观地说,没有一种计算机是真正可以线性关系扩展的。

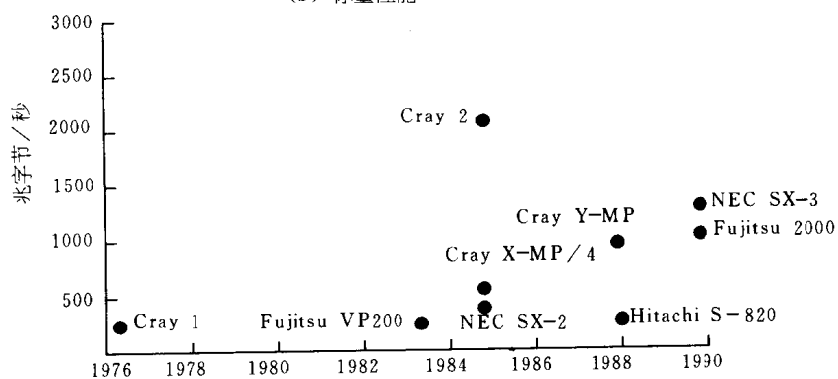
- **换代可扩展性** 由于微处理机每用三年就要过时,所以换代可扩展性和规模可扩展性一样重要。不仅像制造处理机与存储器芯片的 CMOS 电路和组装技术等硬件技术应该是可



(a) 单处理机向量性能



(b) 标量性能



(c) I/O性能

图 6.16 高性能向量处理机性能数据

扩展的,而且那些要求与新硬件系统兼容和可移植的软件/算法也应该是可扩展的。

• **问题可扩展性** 问题规模是指数据集规模。一台问题可扩展的计算机应在问题规模增大时仍能很好地运行,而且问题规模扩展到相当大时计算机仍能高效地运行。

3. 增加存储器系统的容量和性能

过去和当前向量处理计算机的主存储器和扩充存储器容量大小如图 6.17 所示。大规模存储器系统必须为标量处理提供低时延、为向量处理提供高频宽、为解决大型复杂问题提供大容量和高吞吐率的性能。

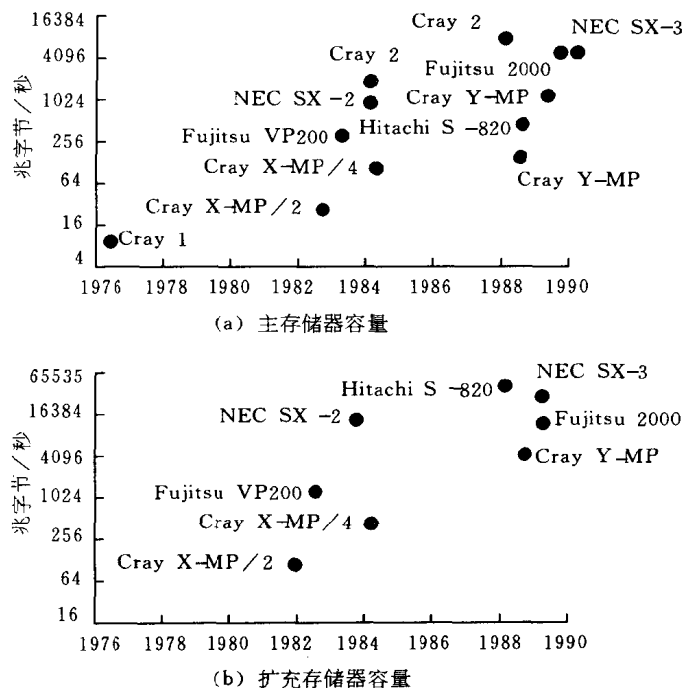


图 6.17 向量处理计算机存储器容量

为了实现以上目标,存储器必须采用高效的层次结构。典型的层次结构由数据文件或磁盘、动态 RAM 的扩充存储器、静态 RAM 的快速共享存储器以及用 RAM 阵列实现的高速缓存/局部存储器组成。

在未来的 10 年中,主存储器容量可望达到 32G 字节,并且可以扩充到 4 倍之大。它将与向量处理计算机系统结构的增长相匹配。

4. 提供高性能的 I/O 和易访问的网络

由于向量处理机每换一代它的速度至少要增加 3~5 倍,因此能够处理的问题规模也相应地增加了,对 I/O 频宽的要求也要提高。图 6.16(c)表示当前和过去向量处理计算机系统所支持的 I/O 频宽。

I/O 是指大型机和外围设备或网络之间的数据传输。老一代向量处理计算机的 I/O 频宽总不能很好地与计算性能相匹配。Cray 公司采用两种不同方法实现 I/O 处理机的系

统结构。采用第一种方法的例子是 Cray Y-MP I/O 子系统, 它的 I/O 处理机非常灵活, 而且可以执行相当复杂的处理工作。Cray 2 采用第二种方法, 用一台简单的前端处理机控制通道, 完成由大型机操作系统执行的 I/O 管理。在未来 5 至 10 年中, 与高速磁盘阵列网络连接的向量处理计算机要求 I/O 传输率能高于 50G 字节/秒。向量处理计算机对高速连网能力的支持将变成 I/O 系统结构的主要组成部分。

6.4.2 提高向量处理机性能的常用技术

1. 链接技术

在寄存器-寄存器系统结构中, 所有的向量操作数在把它们送入流水线之前, 都要预先装入向量寄存器中。中间和最后结果(流水线输出)在把它们存入主存储器以前, 也要把它们装入向量寄存器中。下面, 我们考虑与 Cray 1 那样的寄存器-寄存器向量处理机有关的资源预定问题。向量指令可分成四类, 如图 6.18 所示。

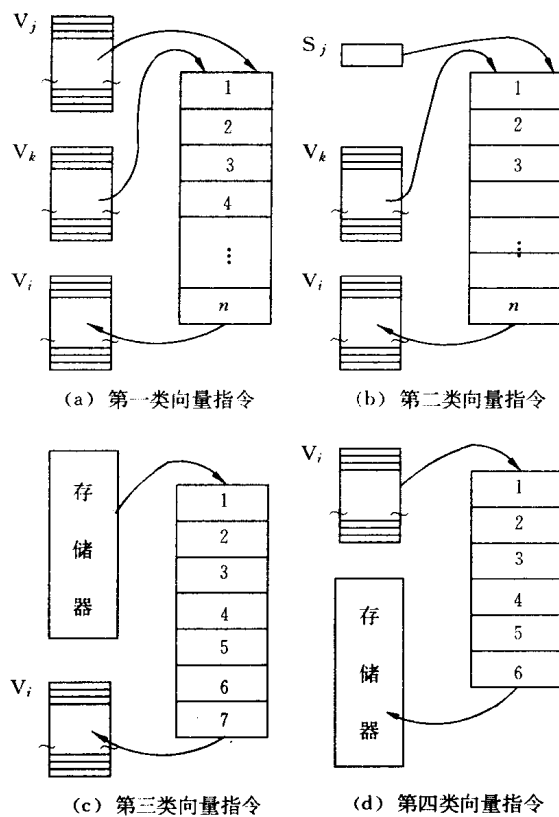


图 6.18 Cray 1 的四类向量指令

第一类指令从一个或二个向量寄存器取得操作数并把结果回送到另一个向量寄存器中去。第二类向量指令从 S_j 寄存器取得一个标量操作数, 又从 V_k 寄存器取得一个向量操

作数,并且把向量结果回送给另一个向量寄存器 V_i 。第三类以及第四类指令分别是把数据从存储器传送到一个向量寄存器中以及反过来把数据从向量寄存器传送到存储器中。存储器和寄存器之间的数据通路可以看成是具有固定时间延迟的数据传送流水线。

当发出向量指令时,所要求的功能流水线和操作数寄存器便要预定若干个时钟周期,其值取决于向量长度。使用同一组功能部件或操作数寄存器的后继向量指令在预定被释放之前是不能发出的。两个或更多的向量指令如果是不相关的,便可以同时使用不同功能流水线和不同的向量寄存器。这种并发的指令能以相继的时钟周期发出。图 6.19(a)表示了两条独立的指令,其中一条使用加法流水线而另一条使用乘法流水线。图 6.19(b)描述了两个独立的向量加法都要求使用加法流水线。当第一条加法指令发出时,加法流水线就被预定了,所以,第二条加法指令就要延迟到加法流水线空闲后才能发出。图 6.19(c)表明两条不同的向量指令共享同一个操作数寄存器 V_1 。第一条加法指令预定操作数寄存器 V_1 ,使乘法指令要延迟到操作数寄存器 V_1 空闲后才能发出。图 6.19(d)说明加法流水线和操作数寄存器 V_1 都被预定的情况。就象操作数寄存器要求预定一样,结果寄存器也需要预定若干个时钟周期,其值取决于向量长度和流水线延迟。这种预定保证了最后结果能正确地传送到结果寄存器。

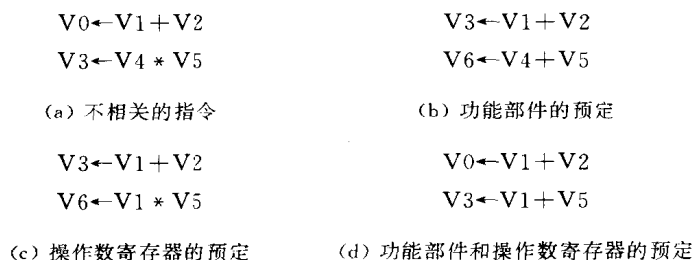


图 6.19 功能部件和操作数寄存器的预定

结果寄存器可能成为后继指令的操作数寄存器。在 Cray 1 中,这种技术称为两条流水线的链接(chaining)。流水线链接是从流水线的内部定向概念发展而来的。链接是当从一个流水线部件得到的结果直接送入另一个功能流水线的操作数寄存器时所发生的连接过程。换句话说,中间结果不必送回存储器,而且甚至在向量操作完成以前就使用。链接允许当第一个结果一变成可用的操作数时就马上发出相继的操作。当然,所需要的功能流水线和操作数寄存器必须恰当地预定;否则,链接操作就不得不挂起直到所需要的资源变为可用为止。下面的例子说明 Cray 1 中的流水线链接。

例 6.1 以下四条向量指令序列链接在一起执行一种复合的功能。

$V_0 \leftarrow$ 存储器 (存储器取)
 $V_2 \leftarrow V_0 + V_1$ (向量加)
 $V_3 \leftarrow V_2 \ll V_3$ (左移)
 $V_5 \leftarrow V_3 \wedge A$ (逻辑积)

图 6.20 给出的图解说明表示把存储器读取流水线、向量加流水线、向量移位流水线

和向量逻辑流水线链接成一个较长的流水线处理器。寄存器 A3 的内容决定移位计数值。

这一链接操作的时间图表示于图 6.21。存储器取数指令是在时间 t_0 发出的。每条水平线表示在寄存器 V5 中一个结果分量的产生。在四条流水线中的时间跨度用粗实线表示(用 b, e, h 和 k 作标记)。虚线表示在存储器取数和功能流水线之间或者是在各个向量寄存器当中的传送之间的通过时间(用 a, c, d, f, g, i, j 和 l 作标记)。每个时钟周期从存储器取一个操作数送到相串联的流水线中去。第一个结果在时钟周期 t_{23} 时形成,此后,每个时钟周期就有一个新的结果分量进入 V5 寄存器。

例 6.2 若要进行向量运算: $D = A \times (B + C)$, 假设向量长度 ≤ 64 , 且 B 和 C 已由存储器取至 V0 和 V1, 则下面 3 条向量指令就可完成上述运算:

V3 ← A

V2 ← V0 + V1

V4 ← V2 * V3

第一、二条指令因既无向量寄存器使用冲突,也无功能部件使用冲突,所以这两条指

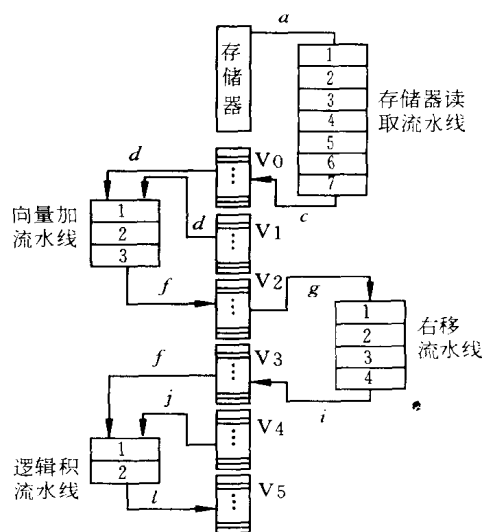


图 6.20 Cray 1 的流水线链接举例

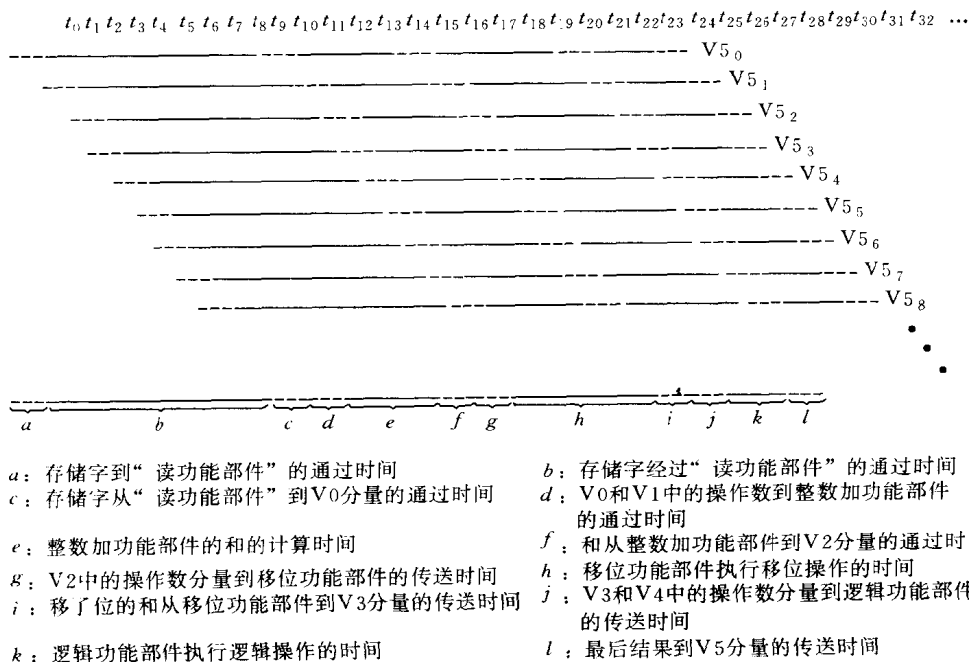


图 6.21 例 6.1 中的链接时间图

令可并行执行。第三条指令与第一、二条指令均存在先写后读的相关冲突,因而可将第三条指令与第一、二条指令链接执行,如图 6.22 所示。

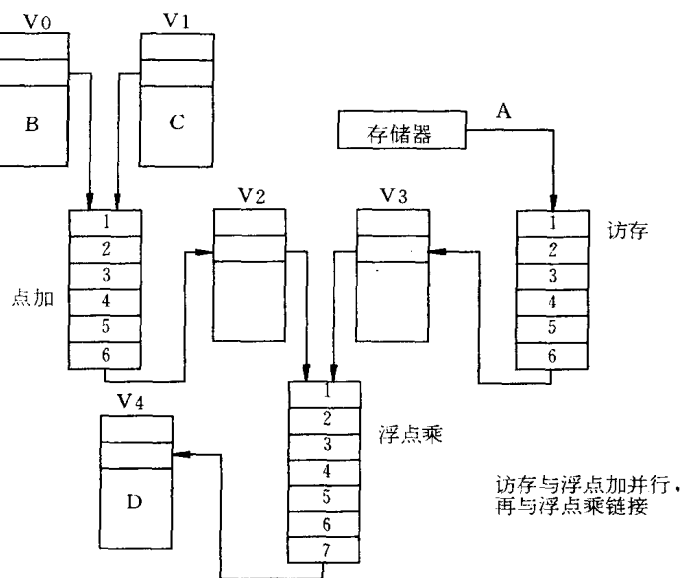


图 6.22 例 6.2 的链接操作

由于同步的要求,数据进入和流出每个功能部件,包括访存都需要 1 拍时间。

假设向量长度为 N , 若这三条指令全部用串行方法,则执行时间为:

$$[(1 + 6 + 1) + N - 1] + [(1 + 6 + 1) + N - 1] + [(1 + 7 + 1) + N - 1] \\ = 3N + 22 \text{ 拍}$$

若前两条指令并行执行,第三条指令串行执行,则执行时间为:

$$[(1 + 6 + 1) + N - 1] + [(1 + 7 + 1) + N - 1] = 2N + 15 \text{ 拍}$$

若采用链接技术,则执行时间为:

$$(1 + 6 + 1) + (1 + 7 + 1) + (N - 1) = 17 + N - 1 = N + 16 \text{ 拍}$$

实现链接除了无向量寄存器使用冲突和无功能部件使用冲突外,还有时间上的要求,只有当前一条指令的第一个结果分量送入结果向量寄存器的那一个时钟周期方可链接,若错过该时刻就不能进行链接,只有当前一条向量指令全部执行完毕,释放向量寄存器资源后才能执行后面指令。另外,当一条向量指令的两个源操作数分别是两条先行指令的结果寄存器时,要求先行的两条指令产生运算结果的时间必须相等,即要求有关功能部件的延迟时间相等,如例中的访存和浮点加功能部件延时均为 6 拍。此外还要求这两条向量指令的向量长度必须相等,否则也不能链接。

2. 向量循环或分段开采技术

当向量的长度大于向量寄存器的长度时,必须把长向量分成长度固定的段。处理长向量的程序结构称为向量循环。这种技术也称为分段开采,一次处理一个向量段。将长向量分段成为循环是由系统硬件和软件控制完成的,程序员看不到这种向量分段为循环的过程,对程序员是透明的。每经过一次循环,便处理长向量的一个段。一般在进入循环以前,

根据向量长度计算出循环计数值。下面是一个说明 Cray 1 实现向量循环的例子。

例 6.3 设 A 和 B 是长度为 N 的向量。考虑以下的循环操作：

DO 10 I=1,N

10 A(I)=5.0 * B(I)+C

当 N 为 64 或更小时,产生 A 数组的 7 条指令序列是:

$S_1 \leftarrow 5.0$	在标量寄存器内设置常数
$S_2 \leftarrow C$	将常数 C 装入标量寄存器
$VL \leftarrow N$	在 VL 寄存器内设置向量长度
$V_0 \leftarrow B$	将 B 向量读入向量寄存器
$V_1 \leftarrow S_1 * V_0$	B 数组的每个分量和常数相乘
$V_2 \leftarrow S_2 + V_1$	C 和 $5 * B(x)$ 相加
$A \leftarrow V_2$	将结果向量存入 A 数组

第 5 条和第 6 条指令使用了不同的功能部件,又共享中间寄存器,它们可以链接在一起,该链的输出最后存入 A 数组。

当 N 超过 64 时,就需要向量循环。在进入循环以前,把 N 除以 64,以确定循环计数值。如果有余数,则在第一次循环中首先产生 A 数组的余数个分量。对于 A 和 B 数组的每 64 个分量的段,循环由第 4 条到第 7 条指令组成。

3. 向量递归技术

在向量操作中,结果通常是不送回到作为源操作数使用的同一个向量寄存器中的。有一类特殊的向量循环,其流水线功能部件的输出可能要回送到它的一个源向量寄存器。换句话说,一个向量寄存器用来同时存放源操作数和结果操作数。在功能流水线上的这种递归操作要求特别小心以避免产生数据阻塞问题。

Cray 机器利用每个向量寄存器的分量计数器实现这一功能。在每个流水线周期,从分量这一角度看,向量寄存器的作用好像移位寄存器。当一个操作数分量移出向量寄存器

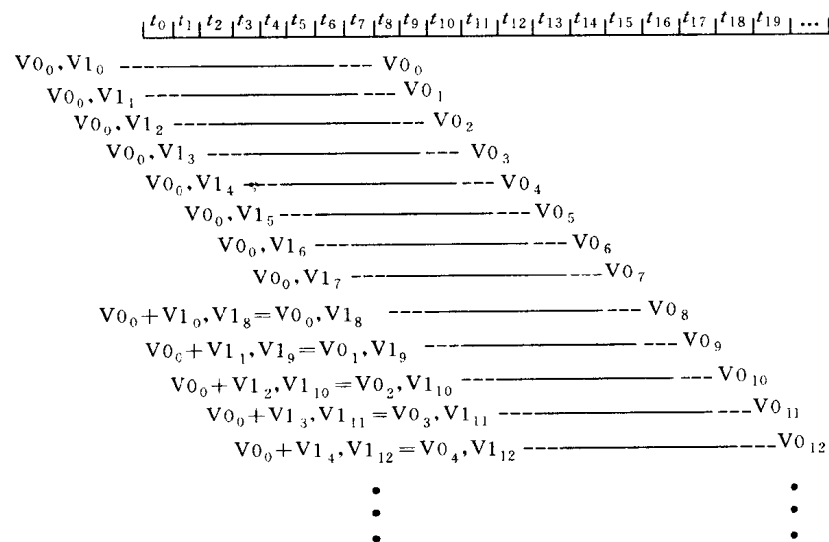


图 6.23 向量分量递归和的时间图

进入流水线功能部件时,一个结果分量可以在同一周期进入腾空的分量寄存器。分量计数器必须跟踪移位操作,直到结果向量的所有 64 个分量都装入向量寄存器。

下面考虑用浮点加法流水线完成递归向量求和 $V_0 \leftarrow V_0 + V_1$, 其中向量寄存器 V_1 保存要进行递归相加的浮点数,向量寄存器 V_0 同时用作操作数寄存器和结果寄存器。令 C_1 和 C_2 分别是与向量寄存器 V_1 和 V_2 相关的分量计数器。初始时,计数器 C_0 和 C_1 都置成 0, V_0 的第一个分量寄存器 V_{0_0} 中的初始值也置成 0。通过浮点加法流水线需要 6 个时钟周期,寄存器和浮点加法流水线需要 6 个时钟周期,寄存器和浮点加法流水线之间的往返传送各还需要 1 个时钟周期,因此,一次加法计算总共需要 $1+6+1=8$ 个时钟周期,如图 6.23 所示。假定向量长度寄存器的值为 64, 只作一个向量循环。

在 t_8 之前,计数器 C_0 一直保持为 0。在此期间, V_{0_0} (为 0) 不断发送到流水线。但是,计数器 C_1 在每个时钟周期后都加 1, 所以,在 t_8 以后,在随后的 64 个时钟周期内 $V_{1_0}, V_{1_1}, \dots, V_{1_{63}}$ 相继发送到流水线。 t_8 以后,每个时钟周期后 C_0 都要加 1。这意味着,在每 8 个时钟周期内,相继输出的和将与来自 V_1 的一个分量进行递归相加。当计算完成时, V_0 的分量寄存器将被装入。64 个分量被分成 8 组, 每组为 8 个分量的和。从 $V_{0_{56}}$ 到 $V_{0_{63}}$ 为最后一个求和的组, 其中每个寄存器保存了 V_1 的 8 个分量的和, 这个组共保存了 8 个这样的求和值。运算结束后, V_0 各个分量的内容如下:

$$\begin{aligned}
 & \left. \begin{aligned} (V_{0_0}) &= (V_{0_0}) + (V_{1_0}) = 0 + (V_{1_0}) \\ (V_{0_1}) &= (V_{0_0}) + (V_{1_1}) = 0 + (V_{1_1}) \\ (V_{0_2}) &= (V_{0_0}) + (V_{1_2}) = 0 + (V_{1_2}) \\ (V_{0_3}) &= (V_{0_0}) + (V_{1_3}) = 0 + (V_{1_3}) \\ (V_{0_4}) &= (V_{0_0}) + (V_{1_4}) = 0 + (V_{1_4}) \\ (V_{0_5}) &= (V_{0_0}) + (V_{1_5}) = 0 + (V_{1_5}) \\ (V_{0_6}) &= (V_{0_0}) + (V_{1_6}) = 0 + (V_{1_6}) \\ (V_{0_7}) &= (V_{0_0}) + (V_{1_7}) = 0 + (V_{1_7}) \end{aligned} \right\} \text{第 1 组} \\
 & \left. \begin{aligned} (V_{0_8}) &= (V_{0_0}) + (V_{1_8}) = (V_{1_0}) + (V_{1_8}) \\ (V_{0_9}) &= (V_{0_1}) + (V_{1_9}) = (V_{1_1}) + (V_{1_9}) \\ (V_{0_{10}}) &= (V_{0_2}) + (V_{1_{10}}) = (V_{1_2}) + (V_{1_{10}}) \\ (V_{0_{11}}) &= (V_{0_3}) + (V_{1_{11}}) = (V_{1_3}) + (V_{1_{11}}) \\ (V_{0_{12}}) &= (V_{0_4}) + (V_{1_{12}}) = (V_{1_4}) + (V_{1_{12}}) \\ &\vdots \\ (V_{0_{15}}) &= (V_{0_7}) + (V_{1_{15}}) = (V_{1_7}) + (V_{1_{15}}) \end{aligned} \right\} \text{第 2 组} \\
 & (V_{0_{16}}) = (V_{0_8}) + (V_{1_{16}}) = (V_{1_0}) + (V_{1_8}) + (V_{1_{16}}) \\
 & \quad \quad \quad \uparrow \\
 & (V_{0_{55}}) = (V_{0_{47}}) + (V_{1_{55}}) = (V_{1_7}) + (V_{1_{15}}) + (V_{1_{55}}) \vdots \text{第 3 组到第 7 组} \\
 & \quad \quad \quad \downarrow \\
 & \quad \quad \quad + (V_{1_{31}}) + (V_{1_{39}}) + (V_{1_{47}}) + (V_{1_{55}})
 \end{aligned}$$

$$\left. \begin{aligned}
 (V_{0_{56}}) &= (V_{0_{48}}) + (V_{1_{56}}) = (V_{1_1}) + (V_{1_8}) + (V_{1_{16}}) + (V_{1_{24}}) \\
 &\quad + (V_{1_{32}}) + (V_{1_{40}}) + (V_{1_{48}}) + (V_{1_{56}}) \\
 (V_{0_{57}}) &= (V_{0_{49}}) + (V_{1_{57}}) = (V_{1_1}) + (V_{1_9}) + (V_{1_{17}}) + (V_{1_{25}}) \\
 &\quad + (V_{1_{33}}) + (V_{1_{41}}) + (V_{1_{49}}) + (V_{1_{57}}) \\
 (V_{0_{58}}) &= (V_{0_{50}}) + (V_{1_{58}}) = (V_{1_2}) + (V_{1_{10}}) + (V_{1_{18}}) + (V_{1_{26}}) \\
 &\quad + (V_{1_{34}}) + (V_{1_{42}}) + (V_{1_{50}}) + (V_{1_{58}}) \\
 (V_{0_{59}}) &= (V_{0_{51}}) + (V_{1_{59}}) = (V_{1_3}) + (V_{1_{11}}) + (V_{1_{19}}) + (V_{1_{27}}) \\
 &\quad + (V_{1_{35}}) + (V_{1_{43}}) + (V_{1_{51}}) + (V_{1_{59}}) \\
 (V_{0_{60}}) &= (V_{0_{52}}) + (V_{1_{60}}) = (V_{1_4}) + (V_{1_{12}}) + (V_{1_{20}}) + (V_{1_{28}}) \\
 &\quad + (V_{1_{36}}) + (V_{1_{44}}) + (V_{1_{52}}) + (V_{1_{60}}) \\
 (V_{0_{61}}) &= (V_{0_{53}}) + (V_{1_{61}}) = (V_{1_5}) + (V_{1_{13}}) + (V_{1_{21}}) + (V_{1_{29}}) \\
 &\quad + (V_{1_{37}}) + (V_{1_{45}}) + (V_{1_{53}}) + (V_{1_{61}}) \\
 (V_{0_{62}}) &= (V_{0_{54}}) + (V_{1_{62}}) = (V_{1_6}) + (V_{1_{14}}) + (V_{1_{22}}) + (V_{1_{30}}) \\
 &\quad + (V_{1_{38}}) + (V_{1_{46}}) + (V_{1_{54}}) + (V_{1_{62}}) \\
 (V_{0_{63}}) &= (V_{0_{55}}) + (V_{1_{63}}) = (V_{0_7}) + (V_{1_{15}}) + (V_{1_{23}}) + (V_{1_{31}}) \\
 &\quad + (V_{1_{39}}) + (V_{1_{47}}) + (V_{1_{55}}) + (V_{1_{63}})
 \end{aligned} \right\} \text{第 8 组 (结果)}$$

上述的递归向量求和在科学计算中是非常有用的。例如,向量的点积 $A \cdot B = \sum a_i \cdot b_i$ 在 Cray 1 中可以用以下两操作的链接来实现向量乘 $V1 \leftarrow V3 * V4$,接着是浮点加 $V0 \leftarrow V0 + V1$ 。如果向量长度是 64,在链接的乘法和加法操作完成后,点积可以从 64 个减少到 8 个和。下一步的迭代是求出 8 个子和的和以产生最后的结果。对于递归向量乘法运算,除 V_{00} 的初始值要设置为 1 而不是 0 外,类似的操作可以用乘法流水线来完成。这种操作在多项式求值中是很有用的。

4. 稀疏矩阵的处理技术

矩阵是许多科学与工程计算问题中研究的数学对象。在许多矩阵问题中非零元素往往是很少的。我们把许多元素的值为零的矩阵称为稀疏矩阵。例如在有限元问题中,经常遇到这种稀疏矩阵,其中非零项表示一个体积元素和相邻体积元素之间的相互作用。非零元素的个数与每个体积元素的相邻体积元素的个数有关。一般说来,非零元素的个数只占整个矩阵项数的很少一部分。

在计算机系统结构方面已经采用了一些方法来解决稀疏矩阵问题。CDC STAR 采用稀疏向量方法。一个稀疏向量由两个向量组成。其中一个是短向量,它仅包含向量的非零元素。另一个是位向量,其中“1”表示对应位置为非零元素,“0”表示对应位置为零元素。位向量的长度与稀疏向量的长度相等。如果向量元素是 64 位的操作数的话,那么现在所需的位数只是原来的 1/64。

当需要访问稀疏向量的时候,CDC STAR 根据位向量来决定对某个特定的单元是否要进行存取。当位向量相应位是零时,就不需要访问了。虽然位向量可以减少访问主存的次数,但是被存取的元素可能在冲突的存储模块中,这就会导致流水线的延迟。这使只访问非零元素而获得的性能提高被抵消掉了。位向量中的零也需一定的额外开销,但比一次完整的存储器访问所需的开销要小得多。

显然,这种类型的系统结构能对稀疏向量进行各种处理。比如把稀疏形式转换成完整的向量形式,或者反过来。流水结构运算器的输入端可以接收稀疏形式的向量,输出端可以输出稀疏形式的向量。

这种方法的主要问题是最好情况下用一位信息来代替 64 位信息。但是大型的稀疏矩阵往往非常稀疏,64 比 1 的节省远远不够。稀疏向量这种方法如何进一步改进仍是一个待研究的问题。

表示稀疏矩阵的另一种方法是只需存储非零元素,另外把非零元素在原始矩阵中的下标值记录在一个数组中。当访问稀疏矩阵元素时需要把下标值转换成存储器的地址。可以采用 Hash 方法把下标转换成地址。如果 Hash 查找发现一个下标,那么表示对应的元素为非零,Hash 表中包含对应元素的存储器地址。如果 Hash 查找失败,表示相应元素为零。采用 Hash 方法存取数据与 Cache 查找十分类似,Cache 查找可以用流水线方法实现,那么 Hash 查找也可以用流水线方法实现。因此,这种处理稀疏矩阵的方法在流水结构的计算机中大有用武之地。

6.5 向量处理机实例

6.5.1 向量处理机的历史与现状

这一节将介绍由美国和日本制造的向量处理机,包括 Cray 系列、CDC/ETA 系列、Fujitsu VP 系列、NEC SX 系列和 Hitachi 820 系列,如表 6.2 所示。

表 6.2 美国和日本制造的向量处理机

系统型号	最大配置、时钟周期、操作系统/编译系统	特色和要点
Cray 1S	有 10 条流水线的单处理机 12.5ns, COS/CF7 2.1	第一台基于 ECL 的超级计算机,1976 年问世
Cray 2S/4-256	256M 字存储器的 4 台处理机,4.1ns, COS 或 UNIX/CF77 3.0	16K 字的本地存储器,移植了 UNIX V,1985 问世
Cray X-MP 416	16M 字存储器的 4 台处理机,128M 字 SSD, 8.5ns, COS CF77 5.0	使用共享寄存器组用于 IPC, 1983 年问世
Cray Y-MP 832	128M 字存储器的 8 台处理机,6ns, CF77 5.0	X-MP 的改进型,1988 年问世
Cray Y-MP C-90	每台处理机 2 条向量流水线,16 台处理机, 4.2ns, UNICOS/CF77 5.0	最大的 Cray 机器,1991 年问世
CDC Cyber 205	有 4 条流水线的单处理机,20ns, 虚拟 OS/FTN200	存储器到存储器系统结构, 1982 年问世
ETA 10E	单处理机,10.5ns, ETAV/FTN 200	Cyber 205 的后继型号,1985 年问世
NEC SX-X/44	每台处理机 4 组流水线,4 台处理机,2.9ns, F77SX	1991 年问世
Fujitsu VP2600/10	5 条流水线的单处理机和双标量处理机, 3.2ns, MSP. EX/F77 EX/VP	使用可重构微向量寄存器和屏蔽,1991 年问世
Hitachi 820/80	512M 字节存储器,18 个流水线功能部件的单处理机,4ns, FORT 77/HAP V23-OC	64 个 I/O 通道,最大传输速率为 288M 字节/秒,1988 年问世

Cray 系列 Cray 1 是 1975 年问世的,它的改进型 Cray 1S 是 1979 年生产的。它是第一台基于 ECL、时钟周期为 12.5ns 的向量计算机。高度流水线和向量处理是这些机器的主要特点。

Cray 1S 的 10 个流水线功能部件可以同时运行,它的计算能力相当于 10 台 IBM 3033 或 10 台 CDC Cyber 7600。最初生产的 Cray 1 使用 Cray 操作系统(COS),配有 Fortran 77 编译器(CF 77 2.1 版),只允许单用户进行批处理。1983 年 Cray X-MP 系列采用了多处理机配置。用 1~4 个 Cray 1 CPU 和共享存储器进行开发。X-MP 的最主要特征是采用共享寄存器组,加快了处理机之间的通信而不必通过共享存储器实现。除了 128 兆字节的共享存储器外,X-MP 系统还有 1G 字节的固存(solid-state storage, SSD)作为扩展的共享存储器,时钟周期也缩短到 8.5ns。当 4 台处理机同时使用 8 条向量流水线进行加和乘时,X-MP-416 的峰值速度可达 840Mflops。继 Cray X-MP 之后于 1988 年生产了 Cray Y-MP,单个系统最多有 8 台处理机,时钟周期为 6ns,共享存储器的容量为 256 兆字节。1990 年生产了 Cray Y-MP C-90,它是一个集成系统,有 16 台处理机,时钟周期为 4.2ns。1985 年推出了 Cray 2S。该系统最多有 4 台处理机,有 2G 字节的共享存储器,超流水线时钟为 4.1ns。Cray 2 的主要贡献是使超级计算机的批处理 COS 改变到多用户 UNIX 系统 V。目前大多数 Cray 向量计算机上的 UNICOS 操作系统是由 UNIX/V 和 Berkeley 4.3BSD 演变而来的。

Cyber/ETA 系列 CDC 公司于 1973 年推出了第一台向量计算机 STAR-100,以后于 1982 年生产了 Cyber 205。Cyber 205 在单处理机配置时最多有 4 条向量流水线,时钟周期为 20ns。与 Cray 和其他向量计算机中采用寄存器到寄存器的系统结构不同,Cyber 205 及其后继的 ETA 10 采用存储器到存储器的系统结构。它的向量指令比较长,其中包含了存储器地址。

最大的 ETA 10 由 8 个 CPU 和共享存储器以及 18 台 I/O 处理机组成。ETA 10 的目标峰值速度是 10Gflops。Cyber 和 ETA 系列机已不再生产,但仍在几个超级计算机中心使用。

日制超级计算机 1991 年 NEC 生产的 SX-X 系列机宣称峰值速度可达到 22Gflops。由 Fujitsu 公司生产的 VP-2000 系列的峰值速度达 5Gflops。这两种机器的时钟周期分别为 2.9ns 和 3.2ns。这些机器的主要特点是采用了共享通信寄存器和可重构的向量寄存器。Hitachi 公司的 820 系列机的峰值速度为 3Gflops。日本超级计算机的研制在高速硬件和交互式向量化编译器方面是较先进的。

NECSX-X44 NEC 宣称这种机器是最快的向量超级计算机(22Gflops 峰值速度),它的系统结构如图 6.24 所示。达到这一性能指标的主要措施之一是使用基于 VLSI 和高密度封装技术的 2.9ns 时钟。

4 台运算处理机通过共享寄存器或通过 2G 字节的共享存储器进行通信。每台处理机有 4 组向量流水线,每组包括 2 条加法/移位流水线和 2 条乘法/逻辑流水线。因此,类似于 C-90,4 台处理机可达到 64 路并行性。

除了向量部件外,还有高速标量部件,它采用了具有 128 个标量寄存器的 RISC 系统结构。指令通过重新排序开发较高的并行性。主存储器为 1024 路的交叉访问存储器。高达 16G 字节的扩展存储器的最大传输率为 2.75G 字节/秒。

系统最多可以配置 4 台 I/O 处理机,每台 I/O 处理机的数据传输率为 1G 字节/秒。

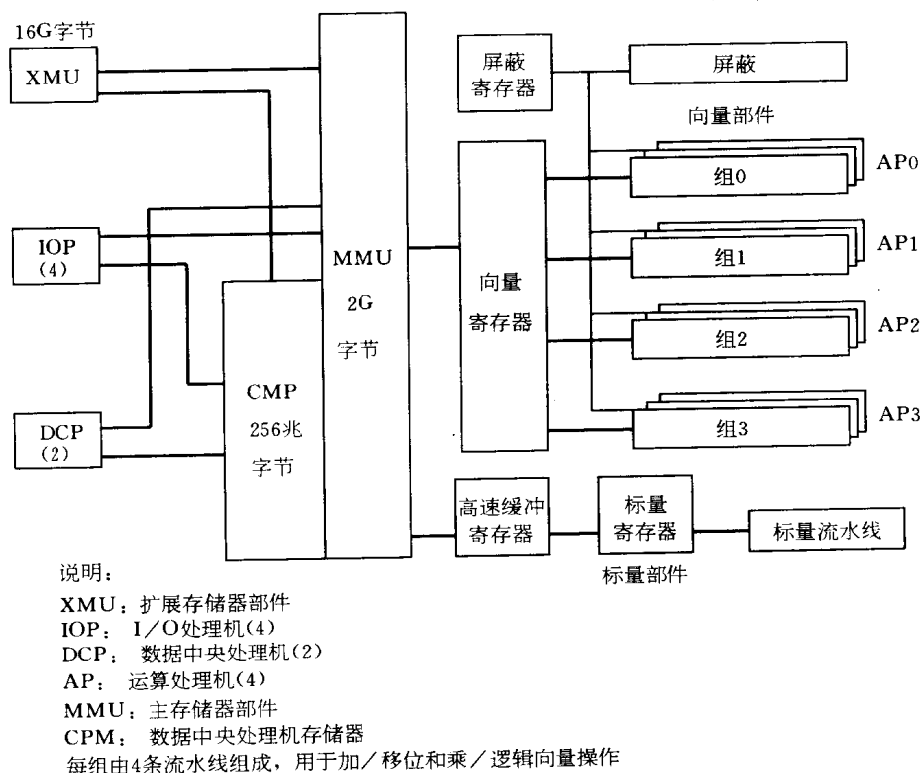


图 6.24 NECSX-X44 向量处理机系统结构

系统最多可以提供 256 个通道用于高速网络、图形和外围操作。系统能支持 100 兆字节/秒的通道工作。

表 6.3 概括了目前三种有代表性的多向量处理机的主要特性。

表 6.3 当前三种向量计算机系统结构的特性

机器特性	CrayY-MP C90/16256	NEC SX-X 系列	Fujitsu VP-2000 系列
处理机台数	16 个 CPU	4 台运算处理机	VP2600/10: 1 台处理机 VP2400/40: 2 台处理机
机器周期时间	4. 2ns	2. 9ns	3. 2ns
最大存储器容量	256 兆字(2G 字节)	2G 字节, 1024 路交叉访问	1G 或 3G 字节的 SRAM
可选的 SSD 存储器	512 兆、1024 兆或 2048 兆字(16G 字节)	16G 字节, 传输速率为 2. 75G 字节/秒	32G 字节的扩充存储器
处理机系统结构: 向量流水线、功能部件和标量部件	每个 CPU 有两条流水线和两个功能部件, 每个时钟周期发送 64 个向量结果	每台处理机有 4 组向量流水线, 每组有两条加法/移位流水线和两条乘法/逻辑流水线, 一条标量流水线	每个向量部件有两条装入/存储流水线和 5 个流水线功能部件。1~2 个向量部件。每个向量部件可以和 2 个标量部件相连

续表

机器特性	CrayY-MP C90/16256	NEC SX-X 系列	Fujitsu VP-2000 系列
操作系统	由 UNIX/V 和 BSD 演变而来的 UNICOS	基于 UNIX 系统 V 和 4.3BSD 的 Super-UX	UPX/M 和用于向量处理的 MSP/EX
前端机	IBM, CDC, DEC, Univac Apollo, Honeywell	内部控制处理器和 4 台 I/O 处理机	与 IBM 兼容的主机
向量化语言/编译器	Fortran 77, C, CF77 5.0, Cray C 3.0 版本	Fortran 77/SX, 向量化器/XS, 分析器/SX	Fortran 77 EX/VP, 有交互式向量化器的 C/VP 编译器
峰值性能和 I/O 频宽	16Gflops 13.6G 字节/秒	22Gflops, 每台 I/O 处理机 1G 字节/秒	5Gflops, 256 个通道, 2G 字节/秒

6.5.2 Cray Y-MP, C-90

下面将介绍 Cray 公司的 Y-MP, C-90 的系统结构。

Cray Y-MP 816 Y-MP 8 系统结构框图如图 6.25 所示。系统可以配置 1 台、2 台、4 台和 8 台处理机。Y-MP 的 8 个 CPU 共享中央存储器、I/O 子系统、处理机通信子系统和实时钟。

中央存储器分成 256 个交叉访问的存储体。通过每个 CPU 对 4 个存储器端口的交叉访问可以实现对存储器的重叠存取。CPU 的时钟周期为 6ns。

中央存储器的容量可以是 16 兆字、32 兆字、64 兆字和 128 兆字,最大可达 1G 字节。固态存储器的容量可以 32 兆字到 512 兆字,最大可达 4G 字节。

4 个存储器访问端口允许每个 CPU 同时执行两个标量和向量取操作、一个存储操作和一个独立的 I/O 操作。这些并行的存储器访问也采用流水线方式,使得向量读和向量写可以同时进行。

系统内部有分解冲突的硬件,使存储器冲突引起的延迟减到最小。为了保护数据,在中央存储器及其输入和输出数据通道中都采用了单错校正/双错检测(SECDED)逻辑。

CPU 的计算系统由 14 个功能部件组成,分为向量、标量、地址和控制四个子系统,如图 6.25 所示。向量和标量指令可以并行地执行。所有算术运算都是寄存器到寄存器类型。向量指令可以使用 14 个中的 8 个功能部件。

系统使用了大量地址寄存器、标量寄存器、向量寄存器、中间寄存器和临时寄存器。通过对寄存器及多条存储器和算术/逻辑流水线的使用,可以实现功能流水线灵活的链接。浮点和整数算术运算都是 64 位。大型指令高速缓存可同时存放 512 条 16 位的指令。

主机中的处理机之间通信系统包括用于快速同步目的的共享寄存器群,每个群由共享地址寄存器、共享标量寄存器和信号灯寄存器组成。CPU 之间向量数据通信是通过共享存储器实现的。

实时钟由 64 位计数器组成,每个时钟周期计数器加 1。由于时钟与程序执行同步,所以它可以用来准确地计算时间。

I/O 子系统支持三类通道,传输速率分别为 6 兆字节/秒,100 兆字节/秒和 1G 字节/秒。

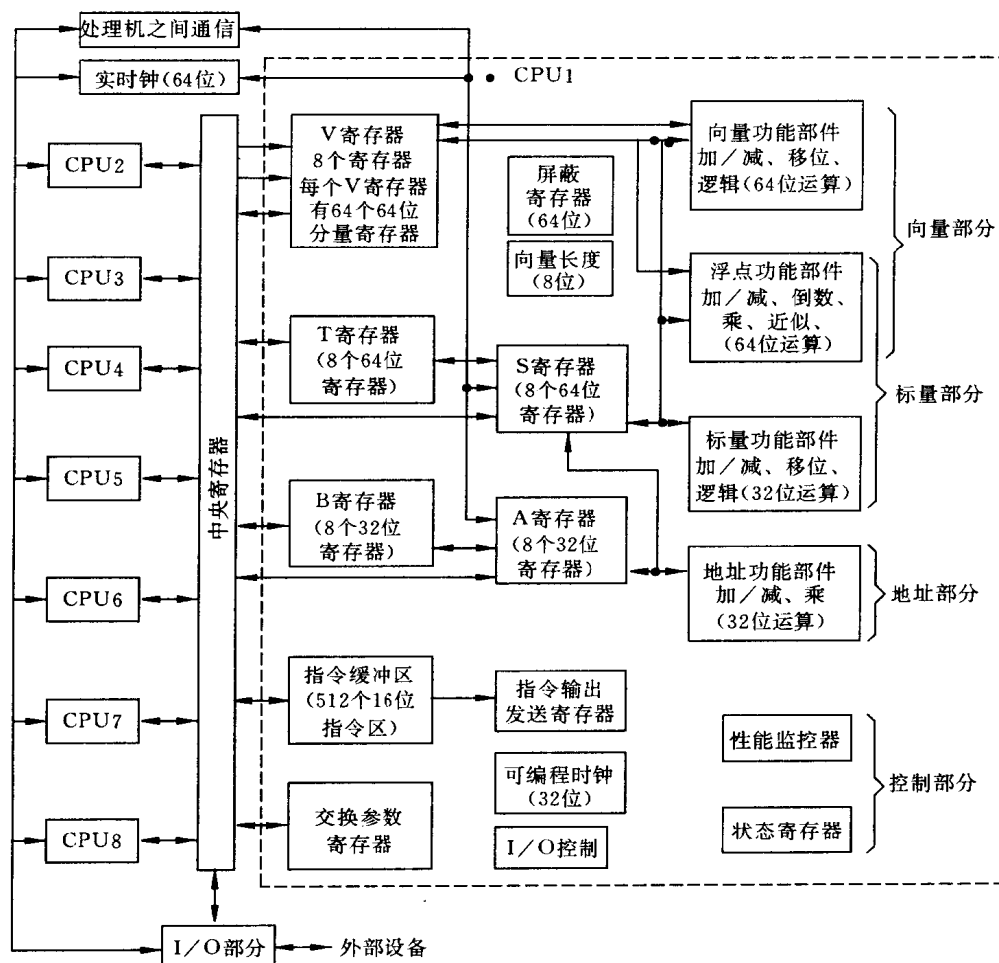


图 6.25 Cray Y-MP 816 系统结构

IOS 和 SSD 是高速数据传输设备,通过 8 个高速缓存支持主机的处理工作。

C-90 和机群 C-90 对 Y-MP 系列在技术上和规模上作了进一步改进。表 6.2 列出了 C-90/16258 系列结构的特征。系统由 16 个类似于 Y-MP 的 CPU 组成。16 台处理机共享主存储的容量高达 256 兆字(2G 字节)。SSD 存储器的容量最长达 16G 字节,可选作第二级主存储器。两条向量流水线和两个功能部件可以并行操作,每个时钟周期能产生 4 个向量计算结果。这意味着每台处理机有 4 路并行性,因此 16 台处理机每个时钟周期最多可以产生 64 个向量计算结果。

C-90 运行 UNICON 操作系统,它是 UNIX 系统 V 和 Berkeley BSD4.3 经过扩充而成的。许多主机可以驱动 C-90。该系统提供向量化的 Fortran 77 和 C 编译器。64 路并行性和 4.2ns 时钟周期相配合,可使系统的峰值性能达到 16Gflops,系统最大 I/O 频宽为 13.6 兆字节/秒。

为了求解大型问题,还可以把多台 C-90 连成机群结构。如图 6.26 所示,4 个 C-90 机

群通过速度为 1 000 兆字节/秒的通道与 SSD 连接。每个 C-90 机群只能访问自己的主存储器,然而它们共享 SSD。换句话说,SSD 中的大量数据供 4 个 C-90 机群共享。每个机群也可以通过共享信号灯部件与其他机群进行通信。只有同步和控制信息才通过信号灯部件传输。在这个意义上说,C-90 机群之间是松散耦合的,但整个系统能提供的最大并行性为 256 路。

如果计算能划分得很好并且机群间负载很均衡,那么配置为 4 个机群的系统时其峰值性能可以达到 64Gflops。Cray Research 实验室在机群结构方面取得了成功的经验。读者可以从 Cray 的报告中直接查得测试程序的结果。

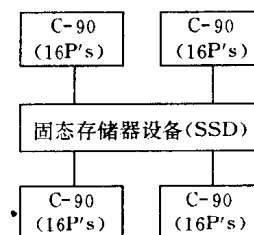


图 6.26 与公共 SSD 相连的四台 Cray Y-MPC-90 构成一个松散耦合的 64 路并行系统

6.5.3 Fujitsu VP2000 和 VPP500

下面介绍 Fujitsu 公司的多向量处理机,VP2000 系列可以配置 1 至 2 台处理机,VPP500 系列的单个 MPP 系统可以配置 7 到 222 个处理部件(PE)。可以联合使用这两系统来求解大型问题。

Fujitsu VP2000 VP-2600/10 单处理机系统的系统结构如图 6.27 所示,它可以扩充成双处理机系统(VP-2400/40)。系统时钟为 3.2ns。主存储器容量为 1G 或 2G 字节。系统存储器可扩充到 32G 字节。

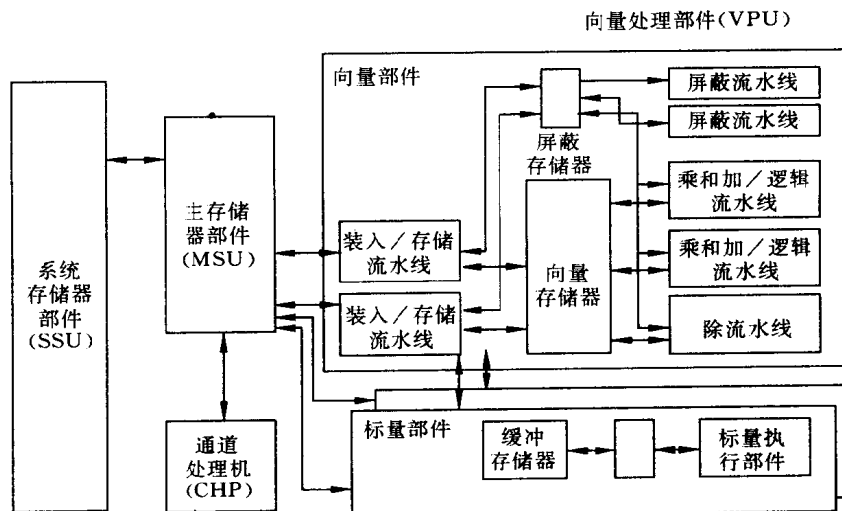


图 6.27 Fujitsu VP2000 系列向量计算机系统结构

每个向量处理部件由二条装入/存储流水线、三条功能流水线和二条屏蔽流水线组成。两个标量部件与一个向量部件相连,因此,在双处理机配置时,最多可以有 4 个标量部件。VP2000 系列有 10 个不同型号的系统,其最大向量性能范围为 0.5 到 5Gflops。

VPP500 这是 Fujitsu 公司向量并行处理机(vector parallel processor)。VPP500 的系统结构可以从 7 个 PE 扩展到 222 个 PE,提供一个高并行的 MIMD 多向量系统。1993 年 9 月问世的第一台系统其目标峰值性能是 335Gflops。图 6.28 是 VP2000 或 VPX200 为主机、VPP500 为后端机的结构示意图。

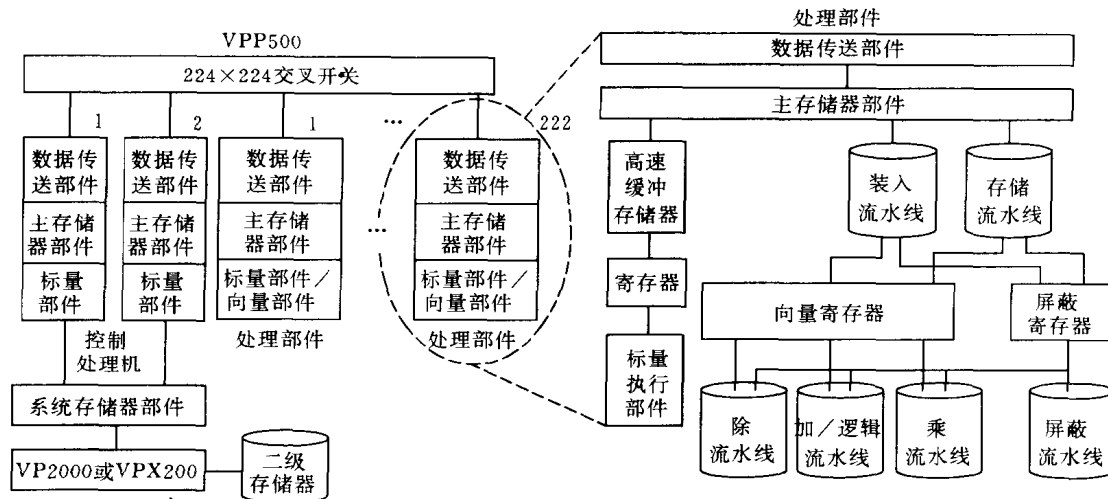


图 6.28 Fujitsu 公司 VPP 500 系统结构

每个 PE 的峰值处理速度为 1.6Gflops,用 256K 个 GaAs 门和 BiCMOS LSI 电路实现。两台控制处理机通过交叉开关网络协调 PE 的操作。每个 PE 中的数据运输部件管理 PE 之间的通信。每个 PE 各自有容量最大可达 256 兆字节的静态 RAM 存储器。系统采用了全局虚拟共享存储器的概念。换句话说,物理上分布的所有 PE 的本地存储器变成一个统一的地址空间。整个系统的主存储器容量最大可达 55G 字节。

每个 PE 有一个标量部件和一个并行操作的向量部件。这些流水线功能部件与 VP2000 非常相似,但流水线的功能已经做过修改,224x224 交叉开关是用于商用 MPP 系统的最大交叉开关。交叉开关网络是无冲突的,在交叉开关阵列的每列中只有一个交叉点接通。

VPP500 和它的主机一起运行基于 UNIX 系统 V 第 4 版的 UXP/VPP 操作系统。该操作系统支持紧耦合的 MIMD 操作。将 FORTRAN 77 编译器的优化功能和基于 UNIX 的操作系统的并行调度功能配合起来则可最大限度地发掘并行向量系统结构的能力。系统采用了 IEEE 754 浮点标准。

每个 PE 的数据传输部件在 PE 间的单向数据交换速度为 400 兆字节/秒,双向数据交换速度为 800 兆字节/秒。该部件将逻辑地址转换成物理地址,以便访问虚拟全局存储器。该部件还配置了专门用于快速栅栏同步的硬件。系统具有可扩展的控制结构。单个控制处理机可以控制多达 9 个 PE。用两台控制处理机就可以协调具有 30 到 222 个 PE 的 VPP 系统。人们希望系统的性能可随 PE 数目的增加而增加,峰值速度达到 11~335Gflops,存储器的容量为 1.8~55 兆字节。

6.5.4 向量协处理器

科学计算要求计算机系统能高速地处理大量数据,向量处理机较好地解决了这一问题,如 CDC STAR, Cray1 向量处理机。但是,这些机器都是巨型机,规模大,价格高,适用于解决大工程和大系统的问题。一般科学计算的用户不可能也不必去购买一台价格昂贵的巨型机,因而只能在中、小型计算机上用标量的处理方式处理向量问题。但是,在这种机器上处理向量问题的效率很低,不能满足这类用户的需要。因此,产生了一种向量协处理器。

向量协处理器是为解决科学计算所要求的大量向量处理而设计的一种装置。它一般和中、小型计算机组合起来,作为主计算机的外围设备,承担处理向量的任务。这样,就可以得到较高的吞吐率和精度,其价格又可以为一般中、小用户所接受。

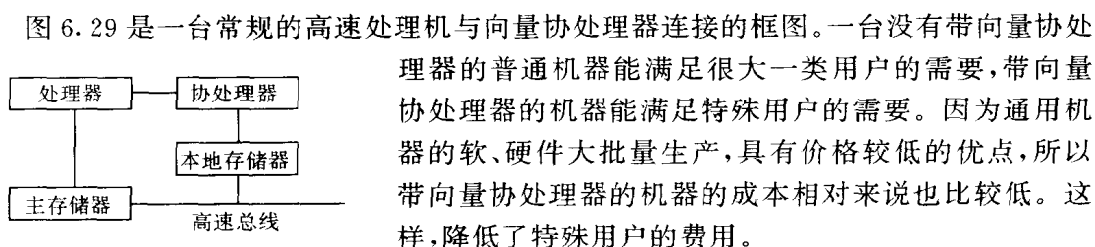


图 6.29 一台带向量协处理器的计算机的结构框图

一台没有带向量协处理器的普通机器能满足很大一类用户的需要,带向量协处理器的机器能满足特殊用户的需要。因为通用机器的软、硬件大批量生产,具有价格较低的优点,所以带向量协处理器的机器的成本相对来说也比较低。这样,降低了特殊用户的费用。

目前一些协处理器制造厂家提供能和各种不同主机相连的向量协处理器。它们的价格和功能的变化范围很大,有和微处理机相连的价格低廉的装置,也有与高档计算机相连的高性能系统。

这一节首先讨论向量协处理器的一般结构。然后介绍美国浮点系统公司的 FPS-164 向量协处理器。

通过前面的讨论我们已经知道,为了实现高效的数值计算,需要有按行和按列等向量存取模式,这要求系统结构设计者设计出能支持这些存取模式的存储器系统。但对算术处理部件的设计没有提出特殊的要求,算术部件的设计也应该支持用户最经常和迫切的运算。因此,让我们复习一下前面遇到过的一些算法。

大多数数值应用的中心问题是求解不同形式的线性方程。线性规划也是通过求解线性方程去解决优化问题。即使是非线性问题,线性技术也十分重要。因为非线性方程组在变化范围较小时呈线性关系,所以通常采用线性迭代方法求解,利用迭代技术通过线性逼近得到非线性系统的解。

对于线性规划和线性代数的操作,计算的内循环经常具有如下的形式:

$$a := a + b \times c \quad (6.1)$$

其中 a, b, c 都是标量。在一台通用的计算机中,先把计算出来的乘积存到某个寄存器,然后将相加的和存到另一个寄存器。由于这种操作非常频繁,我们可以设计一种三个操作数的操作,并假设乘法和加法可在同一运算器上完成,而不需从寄存器中存取乘积,这种结构见图 6.30。其中两个操作数送入乘法

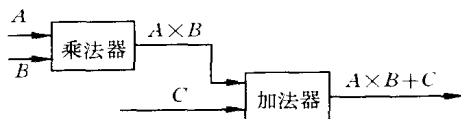


图 6.30 乘法—加法器的结构框图

器,乘法器的输出直接送到加法器的一个输入端,同时第三个操作数送到加法器的另一个输入端。

运算器输入数据的类型不同,式(6.1)的计算情况也不同。如果式(6.1)的输入是向量,输出也是向量,则计算的效率最高。这种情况下,式(6.1)可以表示为:

$$a[1...N] := a[1...N] + b \times c[1...N] \quad (6.2)$$

这个等式的执行过程为:乘法器的一个输入端是标量,另一个输入端是向量 A ,加法器的一个输入端是向量 C ,另一个输入端是乘法器的输出,最后的输出送到为 A 开辟的缓冲存储区。

式(6.1)的另一种计算是两个输入向量通过内积运算变为一个标量。即输入两个向量和输出一个标量。这种情况下,式(6.1)可以表示为:

$$a := a + b[i] \times c[i] \quad (6.3)$$

式(6.3)把 $b[i] \times c[i]$ 的乘积存入标量 a 中。一般的内积运算时 a 的初值为零,但使用式(6.3)的一些算法却要求 a 的初值为非零。实现式(6.3)的一个困难是两个迭代之间需要互锁,因为本次迭代的输出是下一次迭代的输入。如果加法流水线的执行时间是 d 个单位时间,那么两个输出之间需要互锁 $d-1$ 个单位时间,以便使流水线有时间计算出变量 a 的新值供下一次迭代使用。这种方式的执行时间比式(6.2)的执行时间长 d 倍多。互锁使系统的效率大大降低了。

解决这个问题的一个方法是按式(6.4)的顺序计算式(6.3):

$$a_i := a_{i-d} + b_i \times c_i \quad (6.4)$$

按这种方式计算不需要互锁,因为当 a_{i-d} 出现在流水线输出端就可以被流水线输入端引用。

式(6.4)产生了 d 个不同的和,而不是式(6.3)所需的结果,还需要将这 d 个输出变量相加得到最后的结果。最后的求和运算需花费一小段额外时间。当向量 B 和 C 的长度较长时,求和运算的时间可以忽略不计。但是,当向量 B 和 C 的长度较短时,求和运算的时间就不能忽略了。所以当向量长度较短时,应避免采用式(6.4)所示的方法,而应采用式(6.2)所示的方法。

向量协处理器 FPS-164 是当前最典型的向量协处理器。它的结构如图 6.31 所示。

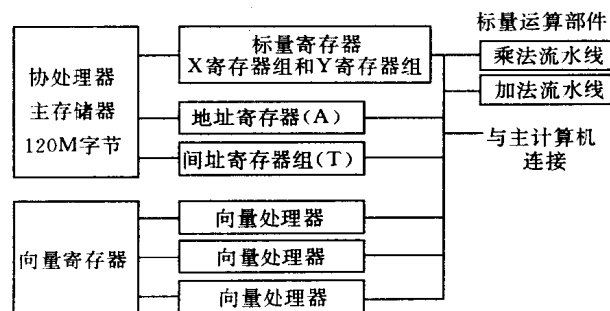


图 6.31 FPS-164 向量协处理器的结构

从图 6.31 可以看到, FPS-164 向量协处理器有自己的主存、高速标量运算器和台数可变的流水结构的向量处理器。整个系统通过高速总线与主计算机相连。主机的功能是为向量协处理器提供数据和程序并接收计算结果。向量协处理器设计成专门高速地处理浮点操作, 一般的应用问题由主计算机来处理。

图 6.31 中的标量处理器适于作快速的标量操作。它包括乘法器、加法器、两组操作数寄存器(X 和 Y 寄存器组)、一组地址寄存器(A 寄存器组)和一组间址寄存器(T 寄存器组)。标量处理器向多达 15 台向量处理器播送指令和数据。图 6.32 是一台向量处理器的框图。

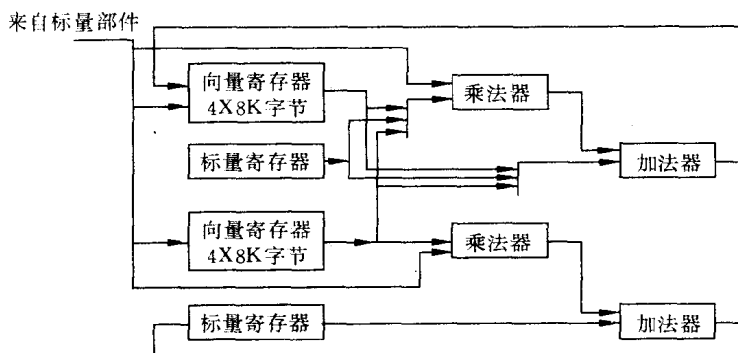


图 6.32 FPS-164 中向量处理器的结构

向量处理器有两个乘-加部件、两组向量寄存器、两组标量寄存器和一个接收标量处理器数据的输入端。每个乘-加部件每个周期能输出一个结果。为了最有效地使用向量处理器, 要求向量处理器有足够的缓冲空间以减少存取向量数据的次数。因此, 向量寄存器的长度为 2K 个操作数, 每个操作数 4 个字节。一组向量寄存器包含 4 个这样的向量寄存器。所以, 一台向量处理器能容纳 $2 \times 4 \times 2K = 16K$ 个向量元素。

每组标量寄存器能容纳 4 个操作数。确定 4 个标量操作数的原因是: 一个向量最多和 4 个不同的标量相乘。因为一个向量一旦装入向量处理器后可被使用 4 次之多, 这样就减少了访问主存的次数。因此, 为了减少访问存储器的次数, 在系统结构设计时, 确定标量寄存器的数目和向量寄存器的大小要采取十分审慎的态度。

相对于标量处理器来说, 向量处理器的工作是被动的。标量处理器以单送方式或播送方式发送指令和数据给向量处理器。所谓播送是标量处理器同时向所有向量处理器发送指令和数据。标量处理器还可以有选择地从向量处理器的寄存器中读取数据。

一般的运算过程是标量处理器先把原始数据按单送方式而不是播送方式装入向量寄存器。然后, 标量处理器把标量数据和指令播送到全部向量处理器。向量处理器就同步地运算, 但它们处理的数据是各不相同的。

当标量处理器以单送方式发送数据和指令时, 除接收的那台向量处理器外其他处理器都处于空闲状态。因此要尽可能避免使用这种工作方式。由于向量寄存器总共可容纳 $15 \times 8 \times 2\,000 = 240\,000$ 个操作数, 两个或更多个大型矩阵可以存放在向量处理器中, 这样就减少了主存和向量寄存器之间数据交换的次数。

向量操作可以和标量处理器中的标量操作同时进行, 这是提高处理效率的基本方法。

这种机器特别适合于大规模的数值处理。FPS-164 结构的一个优点是它的用户可以根据需求购买他们认为合适台数的向量处理器。而且他们可以使用现有的处理机作为主机,而无需专门去设计或开发主机。

6.6 向量处理机的性能评价

衡量向量处理机的性能的主要参数是向量指令的处理时间 T_{vp} 、向量长度为无穷大时的向量处理机的最大性能 P_{∞} 、半性能向量长度 $n_{1/2}$ 和向量方式的工作速度优于标量串行方式工作时所需的向量长度临界值 n_v 等。下面讨论这些参数。

1. 向量指令的处理时间 T_{vp}

在向量处理机上,执行一条向量长度为 n 的向量指令的时间 T_{vp} 可表示为:

$$T_{vp} = T_s + T_{vf} + (n - 1)T_c$$

其中, T_s 为向量流水线的建立时间,它包括向量起始地址的设置、计数器加 1、条件转移指令执行等。 T_{vf} 为向量流水线的流过时间,它是一条指令从开始译码到流过流水线得到第一个结果元素的时间。 T_c 为流水线“瓶颈”段的执行时间。

如果不存在“瓶颈”,每段的执行时间等于一个时钟周期,则上式也可以写成:

$$T_{vp} = [s + e + (n - 1)]\tau$$

其中, s 为向量流水线建立时间所需的时钟周期数, e 为向量流水线流过时间所需的时钟周期数, n 为向量长度, τ 为时钟周期。

下面我们考虑一组向量操作的执行时间。一组向量操作的执行时间主要取决于下面三个因素:向量的长度、向量操作之间是否存在流水功能部件的冲突和数据的相关性。我们把几条能在一个时钟周期内一起开始执行的向量指令称为一个编队。同一个编队中的向量指令一定不存在流水功能部件的冲突和数据的相关性。如果存在这种冲突和相关,需把它们分在不同的编队之中。

例 6.4 下面一组向量操作能分成几个编队? 假设每种流水功能部件只有一个。

LV V1, Rx	;	取向量 x
MULTSV V2, F0, V1	;	向量和标量相乘
LV V3, Ry	;	取向量 Y
ADDV V4, V2, V3	;	加法
SV Ry, V4	;	存结果

解: 第一条指令 VL 为第一个编队。MULTSV 指令因为与第一条 LV 指令相关,所以它们不能在同一个编队中。MULTSV 指令和第二条 LV 指令之间不存在功能部件冲突和数据相关,所以这两条指令为第二个编队。ADDV 指令与第二条 LV 指令数据相关,所以 ADDV 为第三个编队。SV 指令与 ADDV 指令数据相关,所以它为第四个编队。所以这一组向量操作划分为以下四个编队:

- (1) VL
- (2) MULTSV LV
- (3) ADDV

(4) SV

一个编队的执行时间记为 T_{chime} , 它与向量长度无关。因此, 一组由 m 个编队组成的向量操作的执行时间为 m 个 T_{chime} 。如果向量长度为 n , 则整个程序的向量操作的执行时间为 $m \times n$ 个时钟周期。上述例子中, 因为整个程序分为 4 个编队, 所以要花费 4 个 T_{chime} 。另外该例子每个结果需要 2 个浮点运算操作。

除了上述向量操作的真正执行时间外, 还需要考虑向量的启动时间 T_{start} 。 T_{start} 是向量操作流水线的延迟, 它等于流水功能部件的流水段数, 也即流水线的深度。它和上述的向量流水线的流过时间几乎相等。

例 6.5 假设一台向量处理机中功能部件的启动开销为: 取数和存数部件为 12 个时钟周期、乘法部件为 7 个时钟周期、加法部件为 6 个时钟周期。请计算出例 6.4 中每个编队的开始时间、获得第一个结果元素的时间和获得最后一个结果元素的时间。

解: 如果向量长度为 n , 则每个编队的开始时间、获得第一个结果元素和最后一个结果元素时间如表 6.4 所示。

表 6.4 编队(1)~(4)的开始时间、第一个结果和最后一个结果时间

编队	开始时间	第一个结果时间	最后一个结果时间
(1) LV	0	12	$11+n$
(2) MULTSV LV	$12+n$	$12+n+12$	$24+2n$
(3) ADDV	$25+2n$	$25+2n+6$	$31+3n$
(4) SV	$32+3n$	$32+3n+12$	$42+4n$

如果向量长度 n 为 64, 则得到一个结果元素的平均时间为: $4 + (42/64) = 4.65$ 个时钟周期。

• 如果考虑向量长度大于向量寄存器长度时, 则需要分段开采。分段开采的开销由执行标量代码的开销 T_{loop} 和每个编队的向量启动开销 T_{start} 组成。所以向量长度为 n 的一组向量操作的整个执行时间为:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

其中 MVL 是向量寄存器的长度。 T_{start} 、 T_{loop} 和 T_{chime} 的值与编译系统和处理器有关。寄存器分配和指令的调度既会影响编队的组合也会影响每个编队的启动开销。

为了简单起见, 我们把 T_{Tloop} 看作是一个常数, Cray 1 机器的 T_{loop} 约等于 15。

下面我们用例子来说明 T_n 、 T_{start} 和 T_{chime} 的计算。

例 6.6 在一台向量处理机上实现 $A = B \times s$ 操作, 其中 A 和 B 是长度为 200 的向量, s 是一个标量。向量寄存器长度为 64。各功能部件的启动时间和例 6.5 相同。求总的执行时间。

解: 因为向量长度超过了向量寄存器的长度, 所以要采取分段开采方法。每次循环主要由下面三条向量指令组成:

```
LV  V1, Rb      ; 取向量 B
MULTVS V2, V1, Fs ; 向量和标量相乘
```

SV Ra, V2 ; 存向量

这里假设 A 和 B 的最初分别放在 Ra 和 Rb 之中, s 在 Fs 中。其他标量指令这里不一列出了。

这三条指令由于存在相关性,它们必须分属于 3 个编队,因此 $T_{\text{chime}} = 3$ 。根据 T_n 的计算公式,

$$\begin{aligned} \text{则} \quad T_{200} &= 4 \times (15 + T_{\text{start}}) + 200 \times 3 \\ &= 60 + (4 \times T_{\text{start}}) + 600 \\ &= 660 + (4 \times T_{\text{start}}) \end{aligned}$$

其中 T_{start} 是向量取(LV)的启动时间(12 个时钟周期)、向量乘(MULTVS)的启动时间(7 个时钟周期)和向量存(SV)的启动时间(12 个时钟周期)的和,所以 $T_{\text{start}} = 12 + 7 + 12 = 31$,因此 $T_{200} = 660 + 4 \times 31 = 784$ 。一个结果元素的平均执行时间(包括启动开销)为 $784/200 = 3.9$ 。

下面我们举一个向量指令能够链接的例子。

例 6.7 在某台向量处理机上执行 DAXPY(double-precision $a \times X$ plus Y)代码,即完成 $Y = a \times X + Y$,其 X 和 Y 是向量,最初存放在内存。a 是一个标量。它们的向量指令如下:

```
LV V1, Rx          ; 取向量 X
MULTSV V2, F0,V1    ; 标量和向量相乘
LV V3,Ry            ; 取向量 Y
ADDV V4, V2,V3      ; 相加
SV Ry V4            ; 存结果
```

我们可把上述五条向量指令分成三个编队,并且前两个编队中每两条指令组成一条链。

```
LV V1, Rx MULTSV V2, F0, V1    编队 1: 取数和乘法指令链接
LV V3, Ry ADDV V4, V2, V3      编队 2: 取数和加法指令链接
SV Ry, V4                      编队 3: 把结果存入
```

可知 $T_{\text{chime}} = 3$, $T_{\text{loop}} = 15$, $T_{\text{start}} = 12 + 7 + 12 + 6 + 12 = 49$, $MVL = 64$ 。代入 T_n 的计算公式,得:

$$\begin{aligned} T_n &= \left\lceil \frac{n}{MVL} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}} \\ &= \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n \\ &= (n + 64) + 3n \\ &= 4n + 64 \end{aligned}$$

2. 最大性能 R_{∞}

R_{∞} 表示当向量长度为无穷大时的向量流水线的最大性能。常在评价峰值性能时使用,单位为 MFLOPS。它可表示为:

$$R_{\infty} = \lim_{n \rightarrow \infty} \frac{\text{浮点运算次数} \times \text{时钟频率}}{\text{循环所花费的时钟周期数}}$$

因为分子的值与 n 无关, 所以

$$R_{\infty} = \frac{\text{浮点运算次数} \times \text{时钟频率}}{\lim_{n \rightarrow \infty} \text{循环所花费的时钟周期数}} = \frac{\text{浮点运算次数} \times \text{时钟频率}}{\lim_{n \rightarrow \infty} \left[\frac{T_n}{n} \right]}$$

对于例 6.7, 假设时钟频率为 200MHz, 因为每个循环只有 2 次浮点运算操作, 所以

$$R_{\infty} = \frac{2 \times 20\text{MHz}}{\lim_{n \rightarrow \infty} \left[\frac{4n+64}{n} \right]} = \frac{2 \times 200\text{MHz}}{4} = 100\text{MFLOPS}$$

3. 半性能向量长度 $n_{1/2}$

它是为达到一半 R_{∞} 值所需的向量长度。它是评价向量流水线建立时间对性能影响的参数。它表示为建立流水线而导致的性能损失。若向量长度 $n = n_{1/2}$, 则表明整个向量流水处理时间中只有一半时间是在做有效操作, 而另一半是浪费掉的。通常希望向量流水线有较小的 $n_{1/2}$ 。实际测试表示, Cray 1 的 $n_{1/2} = 10 \sim 20$, CYBER 205 的 $n_{1/2}$ 为 100, 表示 Cray 1 的流水线建立时间比 CYBER 205 的要小很多。

现在我们计算在 200MHz 时钟频率的向量处理机上执行 DAXPY 程序程序的 $n_{1/2}$ 。由 MFLOPS 定义可知 MFLOPS:

$$\frac{\text{执行 } n_{1/2} \text{ 循环时的浮点运算次数}}{\text{执行 } n_{1/2} \text{ 循环的时钟周期数}} \times \frac{\text{时钟周期}}{\text{秒}} \times 10^{-6}$$

因为 $R_{\infty} = 100\text{MFLOPS}$, 所以

$$50 = \frac{2 \times n_{1/2}}{T_{n_{1/2}}} \times 200$$

即 $T_{n_{1/2}} = 8 \times n_{1/2}$, 设 $n_{1/2} \leq 64$, 所以 $T_{n_{1/2}} = 1 \times 64 + 3n$

可得 $1 \times 64 + 3 \times n_{1/2} = 8 \times n_{1/2}$

$$5n_{1/2} = 64$$

$$n_{1/2} = 12.8$$

所以 $n_{1/2} = 13$

4. n_v

它表示向量流水方式的工作速度优于标量串行方式工作时所需的向量长度临界值。该参数既衡量建立时间, 也衡量标量、向量速度比对性能的影响。

下面计算上述例子的 n_v 。我们知道 $n_v < 64$ 。如果标量方式工作, 那么一个循环的执行时间大约为 $10 + 12 + 12 + 7 + 6 + 12 = 59$ 个时钟周期, 这里 10 是建立循环的开销, 其他为取数、乘、加、存数等流水线的开销。如果按向量方式工作, 则执行时间为 $T_{n_{1/2}} =$

$$64 + 3 \times n \text{ 时钟周期。因此, } 64 + 3n_v = 59n_v, n_v = \left\lceil \frac{64}{56} \right\rceil, n_v = 2。$$

6.7 关于向量处理机的几点看法

向量计算机系统结构的发展趋势是:

- (1) 提供向量指令；
- (2) 除具有向量处理功能外还有其他功能；
- (3) 采用多层次的存储器系统；
- (4) 流水线技术与并行技术相结合。

向量计算机系统结构是如何解决下面六个技术问题的。

1. 处理器带宽。介绍了两种增加处理器带宽的方法。一是运算器采用流水线结构，另一个是用多个运算器构成完全并行的系统。

2. 存储器带宽。设计者可以用多个彼此独立的存储体构造一个大容量的存储器系统。整个系统的带宽将随着存储体的数目的增加而增加。为了和运算器的带宽匹配，可以采用多层次的存储器系统。其中高速缓冲存储器和可寻址的寄存器组的效果最好。发展的趋势是高速缓冲存储器的容量越来越大，目前的存储容量为 256K~1M。

如果把存储器和运算器看作是常规计算机系统的两个瓶颈，那么流水线结构可使这两个瓶颈得到缓解。对存储器系统来说，如果采用流水线技术，存取一个操作数的速率比传统存储系统快 5~20 倍。同样，流水结构的运算器产生累加和与乘积的速度是传统串行结构运算器的 3~10 倍。采用流水技术，速度的改善很明显，而付出的代价与整个计算机系统的价格相比还是相当低的。因此以流水结构存储器系统驱动的流水结构运算器会不断发展。

3. 输入输出带宽。虽然这一章没有讨论输入输出问题，但如果假定输入输出操作需要一定比例的存储器操作，那么随着存储器带宽的增加，输入输出的带宽也增加了。许多高性能系统都配备 10~20 个直接存储器访问(DMA)通道，其速度与主存的速度相匹配。这些通道依赖于高带宽的存储器，通常通过采用多个存储体获得解决。

4. 通信带宽。本章讨论的大多数向量计算机不需要处理器与处理器之间进行通信。信息分布在向量运算的各操作数之中。例如通过标量点积把两个向量的所有元素的信息结合起来。信息也可以通过公共存储器分布到不同的向量上。算术部件通过访问主存得到不同计算的结果。这种情况下，通信带宽与存储器带宽是同一数值。

5. 同步。对单条流水线来说，运算按进入流水线的顺序执行，所以同步是自动进行的。采用多流水线结构的 FPS 164 用一个控制程序使所有流水线同步地工作。

FPS 164 通过指令流使所有处理器每一步都同步。Cray 1 系统结构采用流水线互锁控制向量操作，以便使不冲突的操作可以并行地执行，相关的操作尽可能链接起来重叠地进行，只要这种重叠不至于产生不正确的结果就行。

6. 多用途。本章讨论的向量计算机解决大多数向量问题都是有效的，主要原因是它们能支持多种存取数据的方式，并且有丰富的向量指令。然而，它们的应用仅仅局限于数值计算问题，对于非数值计算问题是否有效还不大清楚。

通过上面的分析可以看到向量计算机系统结构的主要优点是：

- (1) 通过流水存取方式有效地使用了存储器的带宽。
- (2) 流水结构的运算器有很高的性能价格比。
- (3) 非常简单的机制就能满足通信和同步的要求。

这三个优点把高性能和高效率结合起来了。在标量处理机中，一般执行一条运算指令

可以得到一个运算结果。因此,通常用每秒执行多少条指令 MIPS (million instructions per second) 来衡量机器的运算速度。而向量处理机则完全不同,执行一条向量指令往往可以获得几十个或更多的运算结果。显然,再用上述指标来衡量机器速度就不合适了。在科学计算中,常常用每秒获得多少个浮点运算结果来表示机器速度,以 Mflops (million of floating point per second) 作为测量单位。因为计算机执行的指令,除运算指令外,还有更多的服务性指令,如取数、存数、测试、转移等,所以 Mflops 指标不能直接和标量处理机中所用的 MIPS 相比。在每秒执行多少条指令的速度指标中,是把这些服务性指令都考虑在内的,而在每秒获得多少个浮点运算结果的速度指标中,这些指令却都不考虑在内。一般认为,在标量计算机中,执行一次浮点运算需要 2~5 条指令,平均需 3 条指令。因此,如果要把这两种速度指标放在一起比较的话,那么就应该把 Mflops 乘以一个系数,得出相应的 MIPS。

习 题 六

6.1 解释下列术语:

向量流水处理;分段开采;链接技术;向量和标量的平衡点;向量循环;向量递归;向量流水线流过时间;半性能向量长度;最大性能。

6.2 叙述向量流水处理的主要特点,它与标量流水相比有何不同之处?

6.3 向量流水机的工作方式可分为哪两大类? 它们的主要特点是什么?

6.4 向量的加工方法有哪几种? 各有什么特点? 试从加工速度、需用中间变量等方面加以比较。

6.5 以计算 $A_i = B_i \times C_i (i = 1, 2, \dots)$ 为例,估算一下采用向量方式比采用标量方式其速度提高多少? 假定存储器能供得上所需的指令和数据,其存储周期为 A ns,指令部件可以每拍流出一条分析好的指令,操作部件都采用流水方式,乘和加流水线的功能部件时间都为 4 个时钟周期,忽略置向量参数的时间。

6.6 在 CRAY 1 机上, V 为向量寄存器,设向量长度均为 32, s 为标量寄存器,所用浮点功能执行部件的执行时间分别为:加法需 6 拍,相乘需 7 拍,从存储器读数需 6 拍,求倒数近似值需 14 拍,打入寄存器及启动功能部件(包括存储器)各需 1 拍。问下列各指令组中的哪些指令可以链接? 哪些指令可以并行执行? 试说明其原因并分别计算出各指令组全部完成所需的拍数。

(1) $V0 \leftarrow \text{存储器}$ (2) $V2 \leftarrow V0 * V1$

$V1 \leftarrow V2 + V3$ $V3 \leftarrow \text{存储器}$

$V4 \leftarrow V5 * V6$ $V4 \leftarrow V2 + V3$

(3) $V0 \leftarrow \text{存储器}$ (4) $V0 \leftarrow \text{存储器}$

$V3 \leftarrow V1 + V2$ $V1 \leftarrow 1/V0$

$V4 \leftarrow V0 * V3$ $V3 \leftarrow V1 + V2$

$V6 \leftarrow V4 + V5$ $V5 \leftarrow V3 * V4$

(5) $V0 \leftarrow \text{存储器}$ (6) $V3 \leftarrow \text{存储器}$

$$\begin{array}{ll}
 V1 \leftarrow V2 + V3 & V2 \leftarrow V0 + V1 \\
 V4 \leftarrow V5 * V6 & s0 \leftarrow s2 + s3 \\
 s0 \leftarrow s1 + s2 & V3 \leftarrow V1 * V4 \\
 (7) V3 \leftarrow \text{存储器} & (8) V0 \leftarrow \text{存储器} \\
 V2 \leftarrow V0 + V1 & V2 \leftarrow V0 + V1 \\
 V4 \leftarrow V2 * V3 & V3 \leftarrow V2 * V1 \\
 \text{存储器} \leftarrow V4 & V5 \leftarrow V3 * V4
 \end{array}$$

- 6.7 在 CRAY 1 机上,按链接方式执行下述 4 条向量指令(括号中给出相应功能部件时间),如果向量寄存器和功能部件之间的数据传送需 1 拍,试求此链接流水线的流过时间为多少拍? 如果向量长度为 64,则需多少拍能得到全部结果。

$$\begin{array}{ll}
 V0 \leftarrow \text{存储器} & (\text{存储器取数: 7 拍}) \\
 V2 \leftarrow V0 + V1 & (\text{向量加: 3 拍}) \\
 V3 \leftarrow V2 \ll A3 & (\text{按}(A3)\text{左移: 4 拍}) \\
 V5 \leftarrow V3 \wedge V4 & (\text{向量逻辑乘: 2 拍})
 \end{array}$$

- 6.8 某机有 16 个向量寄存器,其中 $V0 \sim V5$ 中分别存放有向量 A、B、C、D、E、F,向量长度均为 8,向量各元素均为浮点数;处理部件采用二个单功能流水线,加法功能部件时间为 2 拍,乘法功能部件时间为 3 拍。采用类似 CRAY 1 的链接技术,先计算 $(A+B)*C$,在流水线不停流的情况下,接着计算 $(D+E)*F$ 。

- (1) 求此链接流水线的流过时间为多少拍?(设寄存器入、出各需 1 拍)。
- (2) 假如每拍时间为 50ns,完成这些计算并把结果存进相应寄存器,此处理部件的实际吞吐率为多少 MFLOPS?

- 6.9 在 CRAY 1 上计算 $Z = A * (B + C)$,设 A、B、C 都为长度为 128 的向量,并已存放在相应的向量寄存器中,都利用浮点功能部件和链接技术,求完成该计算任务所需的最短时间为多少拍? 其实际吞吐率各为多少 MFLOPS?

- 6.10 若某个向量机其向量方式的执行速率 $R_v = 10\text{MFLOPS}$,标量方式的执行速率 $R_s = 1\text{MFLOPS}$,设 α 是程序中可向量化的百分比。要求:

- (1) 推导该向量机的平均执行速率 R_a 的公式。
- (2) 画出在 $(0,1)$ 范围内, R_a 与 α 的关系图。
- (3) 为使平均执行速率 $R_a = 7.5\text{MFLOPS}$,则 α 应取何值。
- (4) 假定 $R_s = 1\text{MFLOPS}$, $\alpha = 0.7$,则为使 $R_a = 2\text{MFLOPS}$, R_v 应取何值。

- 6.11 在一个向量流水机中,假设向量操作速度为标量操作速度的 10 倍,给定一个原先用标量代码写的程序,为了使执行该程序的加速比分别达到 2、4 和 6,则该程序中有多少百分比的代码可向量化? (提示:运用 Amdahl 定律)

- 6.12 在题 6.11 中,若该程序中有 15%的代码是不能向量化的,例如顺序的 I/O 操作。对于剩下的代码部分,重复问题 6.11 要求达到同样的 3 个加速比 2、4 和 6,则该程序中有多少百分比的代码可向量化?

- 6.13 考虑一个如图 6.33 所示的 4 级加法流水线,其中 X 和 Y 为流水线输入线,Z 为流水线输出线。流水线输出端处有一个寄存器 R,它用来暂存中间结果并在适当时刻

反馈到 S_1 。输入 X 和 Y 分别与输出 R 和 Z 经相应多路开关接到 S_1 和 2 个输入端。假设向量 A 的所有元素以每个周期一个元素的速率,通过输入端 X 送入流水线。现若要计算有 N 个元素的向量 A 的累加和,则最少需要多少个时钟周期? 约定若无操作数输入,就认为是将一个 0 值送入流水线,而且流水线的设置时间可忽略不计。

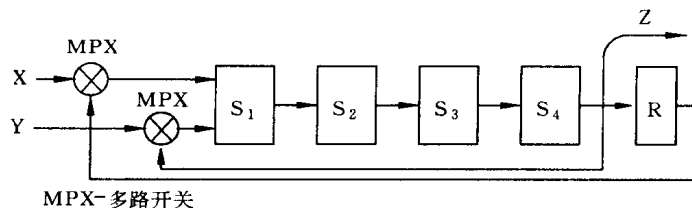


图 6.33 习题 6.13 的附图

- 6.14 设 t 是上题中流水加法器的时钟周期,若现在有一个等效的非流水线加法器,它的流过延迟为 $4t$ 。则当 $N = 64$ 时,计算上述流水对非流水线加法器的加速比 $S(64)$ 以及效率 $E(64)$ 。
- 6.15 在上题中,计算当 N 趋向无限大时的加速比 $S(\infty)$ 和效率 $E(\infty)$ 。
- 6.16 在上题中,计算为达到最大性能的半值,所需的最少向量长度 $N_{1/2}$ 应为多少?
- 6.17 日本 FOCOM 向量处理机 VP-200 具有流水线链接和并行化功能,它有 2 条取存流水线连到向量寄存器,两者可并行工作。此外尚有加法和乘法流水线。各流水线间均可链接操作。现若要在 VP-200 向量机上实现以下的向量操作:
 $A(I) = B(I) * C(I) + D(I) * E(I) + F(I) * G(I)$, 其中 $I = 1, 2, \dots, N$ 。假定所有流水线的延迟时间都相等,取/存操作转换时将有 Δ 的流水线重构延迟时间。试画出完成上述向量操作时,2 条取存流水线、1 条加法和 1 条乘法流水线工作的时空图,并以 N 和 Δ 表示完成操作所需时间。
- 6.18 设 $A(1:3N), B(1:3N), C(1:3N), D(1:3N)$ 为存放在向量机主存中的向量,每个向量都有 $3N$ 个分量,处理机中的每个向量寄存器的长度为 N 个分量。假设该向量机有和题 6.17 中相同的流水线和工作方式,并有足够数量的向量寄存器可供使用,且可串联起来以保存较长的向量如 $2N, 3N$ 等。试计算当执行 $A(I) = [B(I) + C(I)] * D(I)$ 的向量指令串时所需的最小延迟时间,并画出各流水线工作的时空图。

第七章 互连网络

本章讨论多处理机和多计算机中的互连网络,包括互连网络的作用、互连函数、拓扑结构、性能参数、消息寻径机制和互连网络实例等问题。

7.1 互连网络的基本概念

7.1.1 互连网络的作用

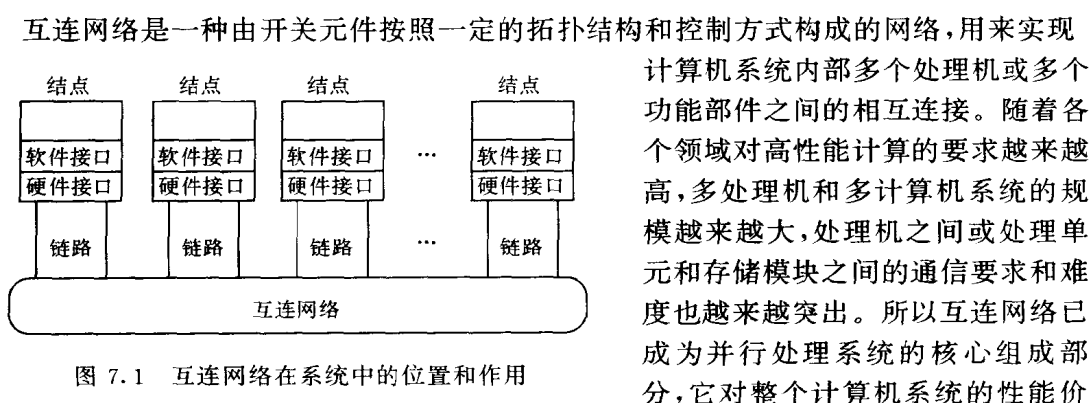
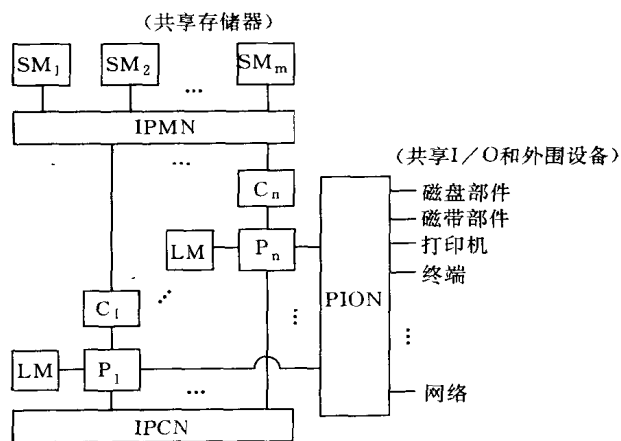


图 7.1 互连网络在系统中的位置和作用

计算机系统内部多个处理机或多个功能部件之间的相互连接。随着各个领域对高性能计算的要求越来越高,多处理机和多计算机系统的规模越来越大,处理机之间或处理单元和存储模块之间的通信要求和难度也越来越突出。所以互连网络已成为并行处理系统的核心组成部分,它对整个计算机系统的性能价格比有着决定性的影响。互连网络在多处理机系统中的位置和作用如图 7.1 所示。

图 7.2 是具有本地存储器、私有高速缓存、共享存储器和共享外围设备的多处理机系



其中: IPMN(处理机-存储器网络);
PION(处理机-I/O网络);
IPCIN(处理机之间通信网络);
P(处理机); C(高速缓冲存储器);
SM(共享存储器); LM(本地存储器)

图 7.2 具有本地存储器、私有高速缓存、共享存储器和共享外围设备的一般处理机系统的互连结构

统互连结构。每台处理机 P_i 与自己的本地存储器(LM)和私有高速缓存(C_i)相连,多处理机-存储器互连网络 IPMN 与共享存储器模块(SM)相连。处理机通过处理机-I/O 网络 PION 访问共享的 I/O 和外围设备。处理机之间通过处理机间通信网络 IPCN 进行通信。

7.1.2 互连函数

为了反映不同互连网络的连接特性,每种互连网络可用一组互连函数来描述。如果将互连网络的 N 个输入端和 N 个输出端分别用整数 $0, 1, \dots, N-1$ 来表示,则互连函数表示相互连接的输出端号和输入端号之间的一一对应关系。或者说,存在互连函数 f ,在它的作用下,输入 i 应与输出 $f(i)$ 相连, $0 \leq i \leq N-1$ 。当互连网络用来实现处理器与处理器之间的数据变换时,互连函数也反映了网络输入数组与输出数组间对应的置换关系或称排列关系。所以互连函数有时也称为置换函数或排列函数。

表示互连函数通常用两种方法:一种是函数表示法,另一种是输入输出对应表示法。

函数表示法用 x 表示输入端变量,用 $f(x)$ 表示互连函数。 x 还常用 n 位二进制形式来表示,写成 $x_{n-1}x_{n-2}\dots x_1x_0$ 。互连函数则对应地表示为 $f(x_{n-1}x_{n-2}\dots x_1x_0)$ 。例如下面将具体介绍的均匀洗牌函数可表示为 $\sigma(x_{n-1}x_{n-2}\dots x_1x_0) = x_{n-2}x_{n-3}\dots x_1x_0x_{n-1}$,左端括号内是输入端变量的二进制表达式, $\sigma(x_{n-1}x_{n-2}\dots x_1x_0)$ 即为均匀洗牌函数,等式右端是该函数的具体表达式。

输入输出对应表示法把互连函数表示为: $\begin{pmatrix} 0 & 1 & \dots & N-1 \\ f(0) & f(1) & \dots & f(N-1) \end{pmatrix}$ 这就是说 0 变换为 $f(0)$, 1 变换为 $f(1)$, \dots , $N-1$ 变换成 $f(N-1)$ 。 f 是互连函数。例如, $N=8$ 均匀洗牌函数的这种表示形式为: $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 2 & 4 & 6 & 1 & 3 & 5 & 7 \end{pmatrix}$ 。有时还把互连函数 $f(x)$ 表示为: $(x_0x_1x_2\dots x_j)$, 所谓“循环”表示,则代表对应关系为: $f(x_0) = x_1, f(x_1) = x_2, \dots, f(x_j) = x_0$, $j+1$ 称为该循环的长度。

下面介绍常用的基本互连函数、它们的函数表达式和主要的特征。

1. 恒等置换

相同编号的输入端与输出端一一对应互连所实现的置换即为恒等置换,其表达式为:

$$I(x_{n-1}x_{n-2}\dots x_1x_0) = x_{n-1}x_{n-2}\dots x_1x_0$$

其中等式左边括号内的 $x_{n-1}x_{n-2}\dots x_1x_0$ 和等式右边的 $x_{n-1}x_{n-2}\dots x_1x_0$ 均为网络输入端和输出端的二进制地址编号。这种置换完成的变换图形如 7.3 所示。图的左部为输入端,右部为输出端。

2. 交换置换

交换置换是实现二进制地址编号中第 0 位位值不同的输入端和输出端之间的连接。其表达式为:

$$E(x_{n-1}x_{n-2}\dots x_1x_0) = x_{n-1}x_{n-2}\dots x_1\bar{x}_0$$

它所实现的输入端与输出端的互连图形见图 7.4。

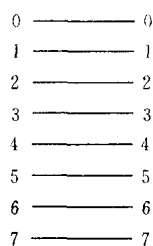


图 7.3 $N = 8$ 的恒等置换

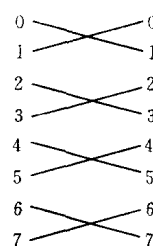


图 7.4 $N = 8$ 的交换置换

3. 方体置换

方体置换是实现二进制地址编号中第 k 位位值不同的输入端和输出端之间的连接。其表达式为:

$$C_k(X_{n-1}X_{n-2}\cdots X_{k+1}X_kX_{k-1}\cdots X_1X_0) \\ = X_{n-1}X_{n-2}\cdots X_{k+1}\bar{X}_kX_{k-1}\cdots X_1X_0$$

这是上述交换置换的一般情形。它应有 C_0, C_1, \dots, C_{n-1} 等 n 个方体置换, 如以 $N = 8$ 为例, 则

$$C_0(x_2x_1x_0) = x_2x_1\bar{x}_0$$

$$C_1(x_2x_1x_0) = x_2\bar{x}_1x_0$$

$$C_2(x_2x_1x_0) = \bar{x}_2x_1x_0$$

其变换图形见图 7.5, 其中 C_0 即为交换置换。

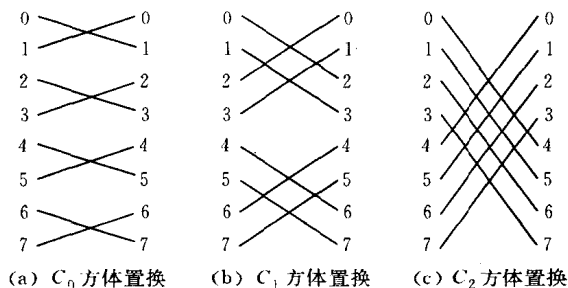


图 7.5 $N = 8$ 的方体置换

4. 均匀洗牌置换

均匀洗牌置换是将输入端分成数目相等的两半, 前一半和后一半按序一个隔一个地从头至尾依次与输出端相连。这好比洗扑克牌时, 将整副牌分成相等的两叠来洗, 以达到理想的一张隔一张的均匀情况, 故称为均匀洗牌置换, 或简称为洗牌置换。其函数关系可表示为:

$$\sigma(x_{n-1}x_{n-2}\cdots x_1x_0) = x_{n-2}x_{n-3}\cdots x_1x_0x_{n-1}$$

由此表达式可见, 洗牌变换是将输入端二进制地址循环左移一位即得到对应的输出端二进制地址。

此外,还可以分别定义子洗牌(subshuffle) $\sigma_{(k)}$ 和超洗牌(supershuffle) $\sigma^{(k)}$ 如下:

$$\sigma_{(k)}(X_{n-1}X_{n-2}\cdots X_{k+1}X_kX_{k-1}\cdots X_1X_0) = X_{n-1}X_{n-2}\cdots X_{k-1}X_kX_{k-1}\cdots X_0X_1X_k$$

$$\sigma^{(k)}(X_{n-1}X_{n-2}\cdots X_{n-k}X_{n-k-1}X_{n-k-2}\cdots X_1X_0) = X_{n-2}\cdots X_{n-k}X_{n-k-1}X_{n-1}X_{n-k-2}\cdots X_1X_0$$

显然成立

$$\sigma^{n-1}(x) = \sigma_{n-1}(x) = \sigma(x)$$

$$\sigma^{(0)}(x) = \sigma_{(0)}(x) = 0$$

图 7.6 示出了 $N = 8$ 的 σ 、 $\sigma_{(2)}$ 和 $\sigma^{(2)}$ 的变换图形。从图可以看出:子洗牌是将整组数据分成若干个子组,对每个子组完成均匀洗牌变换,超洗牌仍对整组数据进行均匀洗牌变换,但增加了数据变换的宽度。

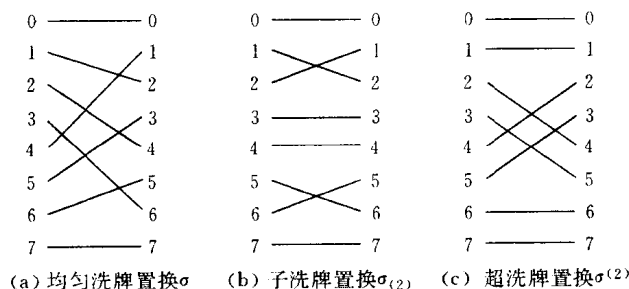


图 7.6 $N = 8$ 的均匀洗牌置换

逆均匀洗牌是均匀洗牌的逆函数,它所完成的变换图形如图 7.7 所示。与图 7.6 相比,两者的输入端和输出端正好互换了个位置,其函数表达式为:

$$\sigma^{-1}(X_{n-1}X_{n-2}\cdots X_1X_0) = X_0X_{n-1}X_{n-2}\cdots X_1$$

这说明逆洗牌是将输入端二进制地址编号循环右移一位即得到相应的输出端地址。

均匀洗牌与逆均匀洗牌是两种十分有用的互连函数,以它们代表的链路与以交换置换代表的开关多级组合起来可构成 Omega (Ω) 网络与逆 Omega (Ω^{-1}) 网络。 σ 函数在实现多项式求值、矩阵转置和 FFT 等并行运算以及并行排序等方面都得到广泛的应用。

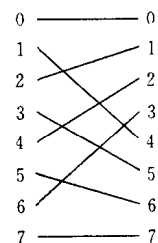


图 7.7 $N = 8$ 的逆均匀洗牌置换

5. 蝶式置换

蝶式置换的名称源于 FFT 变换的实现时其图形的形状如蝴蝶式样。这里,它被定义为:

$$\beta(x_{n-1}x_{n-2}\cdots x_1x_0) = x_0x_{n-2}\cdots x_1x_{n-1}$$

这种置换是将输入端二进制地址的最高位和最低位互换位置即可求得相应输出端的地址。

同样,可定义子蝶式(subbutterfly) $\beta_{(k)}$ 和超蝶式(superbutterfly) $\beta^{(k)}$ 如下:

$$\beta_{(k)}(x_{n-1}x_{n-2}\cdots x_{k+1}x_kx_{k-1}\cdots x_1x_0) = x_{n-1}x_{n-2}\cdots x_{k+1}x_0x_{k-1}\cdots x_1x_k$$

$$\beta^{(k)}(x_{n-1}x_{n-2}\cdots x_{n-k}x_{n-k-1}x_{n-k-2}\cdots x_1x_0) = x_{n-k-1}x_{n-2}\cdots x_{n-k}x_{n-1}x_{n-k-2}\cdots x_1x_0$$

显然,下式也成立

$$\beta^{(n-1)}(X) = \beta_{(n-1)}(X) = \beta(X)$$

$$\beta^{(0)}(X) = \beta_{(0)}(X) = X$$

图 7.8 示出了 $N = 8$ 的 β 、 $\beta_{(2)}$ 和 $\beta^{(2)}$ 的变换图形。蝶式与子蝶式变换与交换变换多级组合可作为构成方体多级网络的基础。

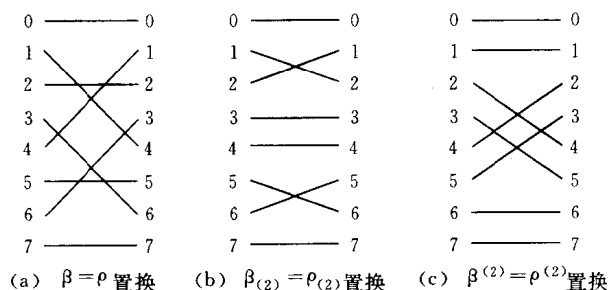


图 7.8 $N = 8$ 的蝶式置换和位序颠倒置换

6. 位序颠倒置换

位序颠倒置换是将输入端二进制地址的位序颠倒过来求得相应输出的地址。其表达式为：

$$\rho(x_{n-1}x_{n-2}\cdots x_1x_0) = x_0x_1\cdots x_{n-2}x_{n-1}$$

同样，可以定义子位序颠倒置换和超位序颠倒置换：

$$\rho^{(k)}(x_{n-1}x_{n-2}\cdots x_{k+1}x_kx_{k-1}\cdots x_1x_0) = x_{n-1}x_{n-2}\cdots x_{k+1}x_0x_1\cdots x_{k-1}x_k$$

$$\rho^{(k)}(x_{n-1}x_{n-2}\cdots x_{n-k}x_{n-k-1}x_{n-k-2}\cdots x_1x_0) = x_{n-k-1}x_{n-k}\cdots x_{n-2}x_{n-1}x_{n-k-2}\cdots x_1x_0$$

对于 $N = 8$ 的情况，正好 $\rho = \beta$ ， $\rho_{(2)} = \beta_{(2)}$ ， $\rho^{(2)} = \beta^{(2)}$ 。其变换图形见图 7.8。但要注意，不要因为这些特殊情况下的 β 和 ρ 的关系而错认为 β 和 ρ 是一样的了。

在实现 FFT 时，最后一步要将结果整序，也就是将结果按二进制位序颠倒重新排列，以获得最后的变换值。

7. 移数置换

移数置换是将输入端数组循环移动一定的位置向输出端传输。其函数表示式无需用二进制编号来写，可表达如下：

$$a(X) = (X + k) \bmod N, \quad 0 \leq X \leq N$$

k 为常数，指移过的位置值，也可以将整个输入数组分成若干个子数组，在子数组内进行循环移数置换，这种段内循环移数的表达式可写成为两个式子如下：

$$a(X)_{(n-1):r} = (X)_{(n-1):r}$$

$$a(X)_{(r-1):o} = ((X)_{(r-1):o} + k) \bmod 2^r$$

其中下标 $(n-1):r$ 和 $(r-1):o$ 分别指从 $n-1$ 位到 r 位和从 $r-1$ 位到 o 位。

循环移数置换的变换图形见图 7.9。这种置换在实现并行计算和图象处理中都很有用。

8. 加减 2^i 置换

加减 2^i 置换实际上也是一种移数置换，其表达式为：

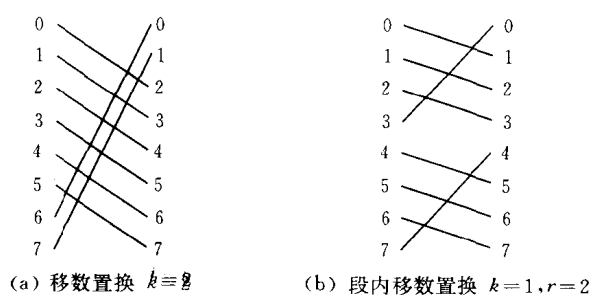


图 7.9 $N=8$ 移数置换

$$PM_{2+i}(X) = X + 2^i \bmod N$$

$$PM_{2-i}(X) = X - 2^i \bmod N$$

其中 $0 \leq X \leq N-1, 0 \leq i \leq n-1, n = \log_2 N$ 。图 7.10 画出这一函数的变换图象。它是构成数据变换网络的基础。

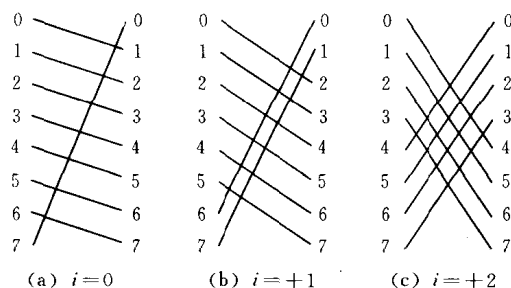


图 7.10 $N=8$ PM2I 置换

Illiacc 函数是构成 Illiac N 阵列的基础,它是 PM2I 函数的一个特例,包含 $PM_{2\pm 0}$ 和 $PM_{2\pm n/2}$ 等四个互连函数。

7.1.3 互连网络的特性和传输的性能参数

1. 互连网络的特性

网络是用有向边或无向边连接有限个结点的图来表示的。下面我们定义几个用于估算复杂性、通信效率和网络价格的参数,即网络特性。

(1) 网络规模 网络中结点数称为网络规模,它表示该网络所能连接的部件多少。

(2) 结点度 与结点相连接的边(即链路或通道)数称为结点度,用 d 表示。在单向通道情况下,进入结点的通道数叫做入度,而从结点出来的通道数则称为出度。结点度就是二者之和。结点度反映了结点所需要的 I/O 端口数,也即反映了结点的价格。为了降低价格,应尽可能使它小。为构造可扩展系统,使构件能模块化,要求结点度保持恒定。

(3) 距离 两结点之间相连的最少边数。

(4) 网络直径 它是网络中任意两个结点之间距离的最大值。它是说明网络通信性能的一个指标。因此从通信的观点来看,网络直径应当尽可能地小。

(5) 等分宽度 当某一网络被切成相等的两半时,沿切口的最小边数(通道)称为通道等分宽度,用 b 表示。于是线等分宽度就是 $B = b \times w$, w 为通道宽度(用位表示)。因此,等分宽度是说明沿等分网络最大通信带宽的一个参数。网络的所有其他横截面都应限在等分宽度之内。

(6) 结点间的线长 它是两个结点间的线的长度。它会影响信号的时延、时钟扭斜和对功率的需要。

(7) 对称性 假若从任何结点看拓扑结构都是一样的话,我们就称此网络为对称网络。对称网络较易实现,编程也较容易。

2. 网络的传输性能参数

下面我们以两台计算机互连的最简单的网络为例讨论网络的传输性能参数。图 7.11 是两台计算机连接的最简单网络。每台计算机有一个 FIFO 的数据队列。

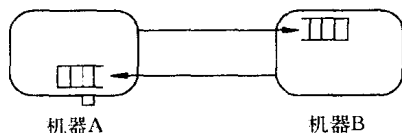


图 7.11 连接两台机器的简单网络

一台机器发送消息给另一台机器时,发送方的步骤如下:

(1) 应用程序把要发送的数据拷贝到操作系统的缓冲区。
(2) 操作系统根据要发送的数据计算出检查和,并把它加在消息中,同时启动超时计数器。

(3) 操作系统把缓冲区中的数据送到网络接口硬件并通知硬件开始发送消息。

消息包的接收和上述步骤正好相反。

(1) 系统把数据从网络接口硬件拷贝到操作系统缓冲区。

(2) 系统根据接收到的数据计算出检查和。如果计算出的检查和与发送过来的检查和匹配,则接收方发一个回答信号给发送方。如果不匹配,则删除这个消息,因为发送方在超时计数器超时后会重发这个消息。

(3) 如果数据通过检查,系统把接收到的数据拷贝到用户地址空间并启动应用程序继续执行。

发送方接收到回答信号后,可以释放系统缓冲区的消息了。如果发送方的超时计数器已超时,那么它重发消息。

下面我们给出互连网络传输方面的性能参数:

(1) 频宽(bandwidth): 它是指消息进入网络后,互连网络传输信息的最大速率。它的单位是兆位/秒,而不用兆字节/秒。

(2) 传输时间(transmission time): 消息通过网络的时间,它等于消息长度除以频宽。

(3) “飞行”时间(time of flight): 消息的第一位信息到达接收方所花费的时间,它包

括由于网络中转发或其他硬件所起的时延。

(4) 传输时延(transport latency): 它等于“飞行”时间和传输时间之和。它是消息在互连网络上所花费的时间,但不包括消息进入网络和到达目的结点后从网络接口硬件取出数据所花费的时间。

(5) 发送方开销(sender overhead): 处理器把消息放到互连网络的时间,这里包括软件和硬件所花费的时间。

(6) 接收方开销(receiver overhead): 处理器把到达的消息从互连网络取出来的时间,这里包括软件和硬件所花的时间。

所以一个消息的总时延可以用下面公式表示:

$$\text{总时延} = \text{发送方开销} + \text{“飞行”时间} + \frac{\text{消息长度}}{\text{频宽}} + \text{接收方开销}$$

这几个性能参数的关系如图 7.12 所示。

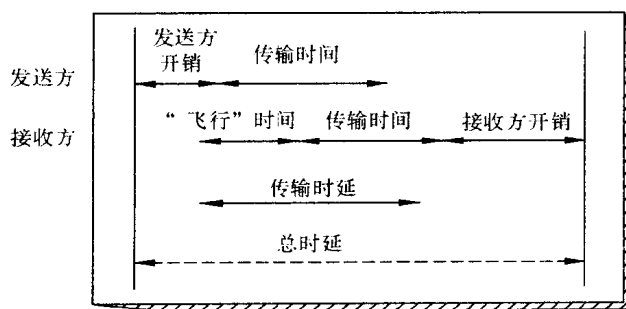


图 7.12 互连网络的传输性能参数

例 7.1 假设一个网络的频宽为 10 兆位/秒,发送方开销和接收方开销分别等于 230 微秒和 270 微秒。如果两台机器相距 100 米,现在要发送一个 1000 字节的消息给另一台机器,试计算总时延。如果两台机器相距 1 000 公里,那么总时延为多大?

解: 光的速度为 299 792.5 公里/秒,信号在导体中传递的速度大约是光速的 50%,所以“飞行”时间可以计算出来了。那么相距 100 米时总时延为:

$$\begin{aligned} T &= \text{发送方开销} + \text{“飞行”时间} + \frac{\text{消息长度}}{\text{频宽}} + \text{接收方开销} \\ &= 230\mu\text{s} + \frac{0.1\text{Km}}{0.5 \times 299\,792.5\text{Km/s}} + \frac{1\,000 \times 8 \text{ 位}}{10 \text{ 兆位/s}} + 270\mu\text{s} \\ &= 230\mu\text{s} + 0.67\mu\text{s} + 800\mu\text{s} + 250\mu\text{s} \\ &= 1301\mu\text{s} \end{aligned}$$

相距 1 000 公里时的总时延为:

$$\begin{aligned} T &= 230\mu\text{s} + \frac{1\,000 \times 10^6}{0.5 \times 299\,792.5} \mu\text{s} + \frac{1\,000 \times 8}{10} \mu\text{s} + 270\mu\text{s} \\ &= 230\mu\text{s} + 6\,671\mu\text{s} + 800\mu\text{s} + 270\mu\text{s} \\ &= 7\,971\mu\text{s} \end{aligned}$$

7.1.4 互连网络的种类

互连网络可分为静态互连网络和动态互连网络两大类。

1. 静态互连网络

静态互连网络是指在各结点间有专用的连接通路,且在运行中不能改变的网络。在静态互连网络中,每一个开关元件固定地与一个结点相连,建立该结点与邻近结点之间的连接通路,直接实现两结点之间的通信。这种网络比较适合于构造通信模式可预测或可用静态连接实现的计算机。一维的静态互连网络有线性阵列结构;二维的有环形、星形、树形、网格形等;三维的有立方体等;三维以上的有超立方体等。下面我们根据网络参数以及它们对网络通信和可扩展性的影响来介绍静态互连网络的拓扑结构。

(1) **线性阵列** 这是一种一维网络,其中 N 个结点用 $N - 1$ 条链路连成一行(图 7.13(a))。内部结点的度为 2,端结点的度为 1。直径为 $N - 1$, N 较大时,直径就比较大。等分宽度 $b = 1$ 。线性阵列是最简单的拓扑结构。这种结构不对称,当 N 很大时,通信效率很低。

在 N 很小的情况下,如 $N = 2$,实现线性阵列是相当经济的。由于直径随 N 线性增大,因此当 N 比较大时,就不应使用这种方案了。应当指出,线性阵列与总线的区别是很大的,后者是通过切换与其连接的许多结点来实现时分特性的。线性阵列允许不同的源结点和目的结点对并发使用系统的不同部分(通道)。

(2) **环和带弦环(chordal ring)** 用一条附加链路将线性阵列的两个端结点连接起来即可得到环(图 7.13(b))。环可以是单向工作,也可以是双向工作。它是对称的,结点度是常数,为 2。双向环的直径为 $N/2$,单向环的直径是 N 。

IBM 令牌环(token ring)的拓扑结构将消息沿环循环传送,直到它们到达有匹配令牌的目的为止。流水线环和包交换环已经在 CDC Cyberplus 多处理机和 KSR-1 计算机系统中实现,用于完成处理机间的通信。

如果将结点度由 2 提高至 3 或 4,我们即可得到如图 7.13(c)和 7.13(d)所示的两种带弦环。分别增加一条和两条附加链路也可以得到这两个带弦环。总之,增加的链路愈多,则结点度愈高,网络直径愈小。

16 个结点的环(图 7.13(b))与两个带弦环(图 7.13(c)和 7.13(d))相比,网络直径分别由 8 减至 5 和 3。在极端情况下,图 7.13(f)全连接网(completely connected network)的结点度为 15,直径最短,为 1。

(3) **循环移数网络** 图 7.13(e)所示的是一个循环移数网络(barrel shifter),其结点数 $N = 16$,它是将环上每个结点到与其距离为 2 整数幂的结点之间增加一条附加链而构成的。这就是说,如 $|j - i| = 2^r, r = 0, 1, 2, \dots, n - 1$,网络规模 $N = 2^n$,则结点 i 与结点 j 连接。这种循环移数网络的结点度为 $d = 2n - 1$,直径 $D = n/2$ 。

显然,循环移数网络的连接特性与结点度较低的任何带弦环相比是有了改进。对 $N = 16$ 的情况,循环移数网络的结点度为 7,直径为 2。但是它的复杂性仍比全连接网络(图 7.13(f))低得多。

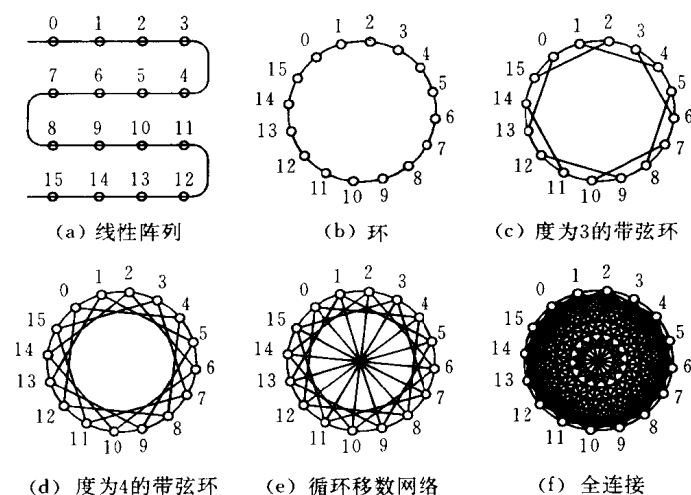


图 7.13 线性阵列、度为 3 和 4 的带弦环、循环移数网络 and 全连接网络

(4) **树形和星形** 一棵 5 层 31 个结点的二叉树如图 7.14(a) 表示。一般说来, 一棵 k 层完全平衡的二叉树应有 $N = 2^{k-1}$ 个结点。最大结点度是 3, 直径是 $2(k-1)$ 。由于结点度是常数, 因此二叉树是一种可扩展的系统结构, 但其直径相当长。

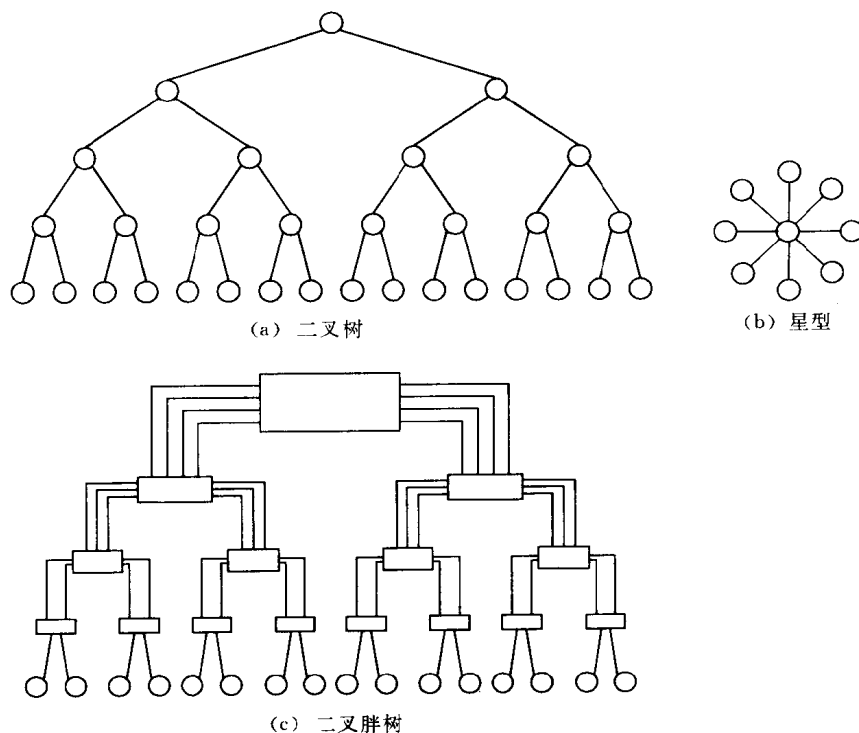


图 7.14 树形、星形和胖树形网络

星形是一种 2 层树, 结点度较高, 为 $d = N - 1$ (图 7.14(b)), 直径较小, 是一常数 2。

哥伦比亚大学研制成的 DADO 多处理机即采用 10 层二叉树形式,有 1023 个结点。星形结构已用于有集中监督结点的系统中。

(5) **胖树形** 1985 年 Leiserson 提出将计算机科学中所用的一般树结构修改为胖树形(fat tree)。二叉胖树结构如图 7.14(c)所示,胖树的通道宽度从叶结点往根结点上行方向逐渐增宽,它更像真实的树,其向树根方向的枝杈变得愈来愈粗。

使用传统二叉树的主要问题之一就是通向根结点的瓶颈问题,这是因为根部的通信最忙。胖树的提出使问题得到了缓解。连接机 CM-5 已采用胖树结构。我们还可将二叉胖树的思想推广到多路胖树。

(6) **网格形和环网形** 图 7.15(a)所示为一个 3×3 网格形网络。这是一种比较流行的结构,它已经以各种变体形式在 Illiac IV、MPP、DAP、CM-2 和 Intel Paragon 中得到了实现。

一般说来, $N = n^k$ 结点的 k 维网格的内部结点度为 $2k$,网格直径为 $k(n-1)$ 。必须指出,图 7.15(a)所示的纯网格形不是对称的。边结点和角结点的结点度为 3 或 2。

图 7.15(b)画出了一种可回绕连接的网格图。假定 Illiac IV 系统采用 8×8 的这种 Illiac 网格,则其结点度为常数 4,直径为 7。 $N = 16 = 4 \times 4$ 构型的 Illiac 网格与图 7.13 所示的结点度为 4 的带弦环在拓扑上是等效的。

一般说来,一个 $n \times n$ Illiac 网格的直径应为 $d = n - 1$,它仅为纯网格直径的一半。图 7.15(c)所示的环形网可看做是另一种直径更短的网格。这种拓扑结构将环形和网格组合在一起,并能向高维扩展。

环网形沿阵列每行每列都有环形连接。一般说来,一个 $n \times n$ 二元环网的结点度为 4,直径为 $2\lceil n/2 \rceil$ 。环网是一种对称的拓扑结构,所有附加的回绕连接可使其直径较之网格结构减少二分之一。

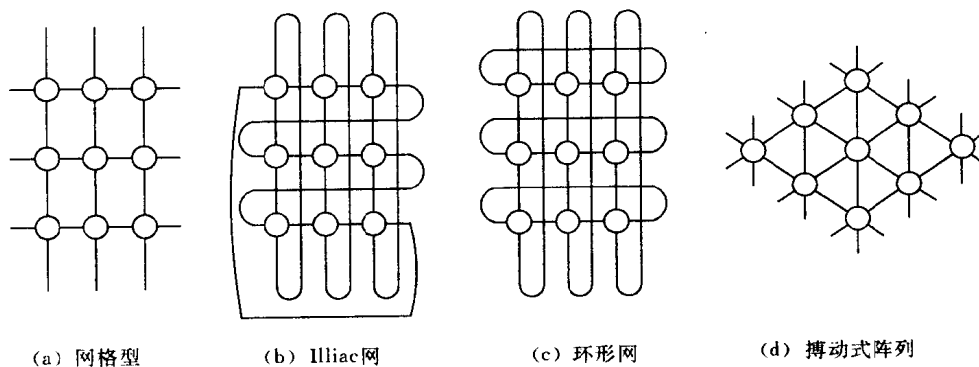


图 7.15 网格形、Illiac 网、环形网和抖动式阵列

(7) **抖动式阵列** 这是一类为实现确定算法而设计的多维流水线阵列结构。图 7.15 (d)所示就是为完成矩阵-矩阵相乘而专门设计的抖动式阵列。此例的内部结点度为 6。

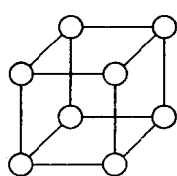
一般说来,静态抖动式阵列可在多个方向上使数据流变成以流水线方式工作。商用 Intel iWarp 系统就是用抖动式结构设计而成的。自从 1978 年 Kung 和 Leiserson 提出抖动式阵列后,它已成为广泛研究的领域。

通过确定的互连和同步操作,搏动式阵列可与算法的通信结构相匹配。对信号/图象处理等特殊应用,搏动式阵列可提供更好的性能/价格比。但是,其结构的实用性有限,而且编制程序也很难。

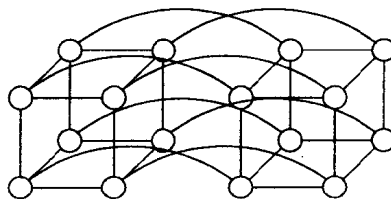
(8) **超立方体** 这是一种二元 n -立方体结构,它已在 iPSC、nCUBE 和 CM-2 系统中得到实现。一般说来,一个 n -立方体由 $N = 2^n$ 个结点组成,它们分布在 n 维上,每维有两个结点。8 个结点的 3-立方体如图 7.16(a)所示。

4-立方体可通过将两个 3-立方体的相应结点互连组成。如图 7.16(b)所示。一个 n -立方体的结点度等于 n ,也就是网格的直径。实际上,结点度随维数线性地增加,所以很难设想超立方体是一种可扩展结构。

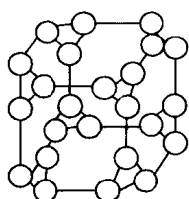
在 80 年代的研究开发工作中,二元超立方体是一种极为普遍采用的结构。Intel iPSC/1、iPSC/2 和 nCUBE 机都是用超立方体结构制造的,这种结构的连接比较密集。许多其他结构诸如二叉树、网格形等都能嵌入超立方体。



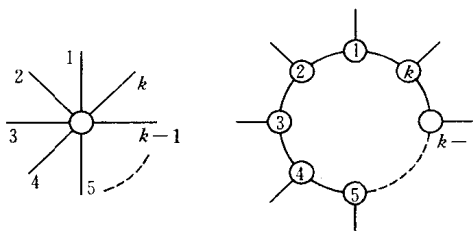
(a) 3-立方体



(b) 由 2 个 3-立方体组成的 4 立方体



(c) 带环 3 立方体



(d) 将 k -立方体的每个结点用 k 个结点的环来代替,组成带环 k -立方体

图 7.16 超立方体和带环立方体

由于缺乏可扩展性以及难于组成高维超立方体,所以超立方体结构正逐渐被其他结构所取代。例如,CM-5 选用了胜过 CM-2 超立方体的胖树形结构。Intel Paragon 选用了较其前身超立方体更好的 2 维网格形结构。拓扑等效已在许多网络结构之间得到证实。一种结构能在未来的系统中继续生存下去的基础是它能够模块地增长,以便高效组装起来,具备可扩展性。

(9) **带环立方体** 这种结构是从超立方体改进而来的。如图 7.16(c)所示,一个 3-立方体可改成带环 3-立方体(CCC)。构成的办法是将 3-立方体的角结点(顶角)用一个结点环来代替。

一般说来,我们可以从一个 k -立方体构成一个有 $n = 2^k$ 个结点环的带环 k -立方体,如图 7.16(d)所示。所用的办法是用 k 个结点的环取代 k 维超立方体的每个顶角。这样,一

个 k -立方体可变成 $k \times 2^k$ 个结点的 k -CCC。

·图 7.16(c)所示 3-CCC 的直径为 6,比原来 3-立方体的直径大一倍。一般说来, k -CCC 的网格直径为 $2k$ 。CCC 的主要改进之处即在其结点度为常数 3,与超立方体的维数无关。

假设一超立方体有 $N = 2^n$ 结点。一个有同样 N 结点数的 CCC 一定是由低维 k -立方体组成,即 $2^n = k \times 2^k$,其中 $k < n$ 。

例如,对应于 $n = 6$ 和 $k = 4$ 的情况,一个 64 结点的 CCC 可用 4 结点的环取代 4-立方体的角结点组成。CCC 的直径为 $2k = 8$,比 6-立方体的 6 要长些。但是,CCC 的结点度为 3,比 6-立方体的结点度 6 要小。在这层意义上来说,如果容许一定的时延,则 CCC 是一种构造可扩展系统的较好的结构。

(10) **k 元 n -立方体网络** 环形、网格形、环网形、二元 n -立方体(超立方体)和 Ω 网络都是 k 元 n -立方体网络系列的拓扑同构体。图 7.17 所示就是一种 4 元 3 立方体网络。

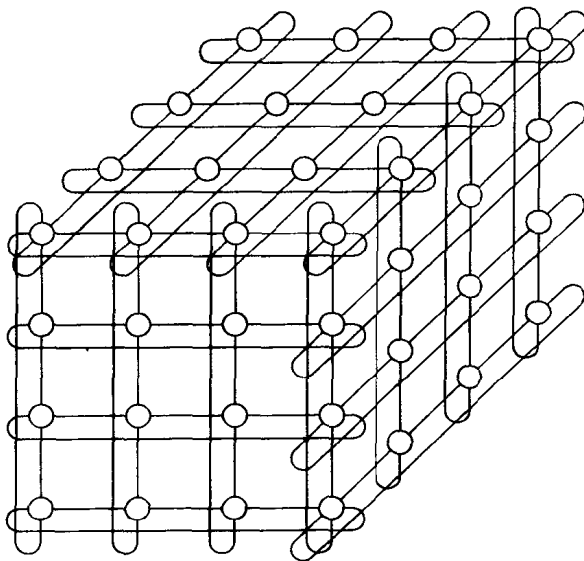


图 7.17 $k = 4$ 和 $n = 3$ 的 k 元 n -立方体网络

参数 n 是立方体的维数, k 是基数或者说是沿每个方向的结点数(多重性)。这两个数与网络中结点数 N 的关系为:

$$N = k^n, \quad (k = \sqrt[n]{N}, n = \log_k N)$$

k 元 n -立方体的结点可用基数为 k 的 n 位地址 $A = a_1 a_2 \cdots a_n$ 来表示,其中 a_i 代表第 i 维结点位置。为简单起见,所有链路都认为是双向的。网络中每条线代表两个通信通道,每个方向一个。图 7.17 中各结点之间的连线都是双向链路。

按照惯例,低维 k 元 n -立方体称为环网,而高维二元 n -立方体则称为超立方体。用网络折叠的方法可避免环网中长的端绕连接,如图 7.18 所示。在此情况下,当多维网络装入一个平面时,每个维上沿环的所有链路的线长相等。这种网络的价格取决于连线量,而不是所需的开关数。在线等分为常数的前提下,宽通道低维数网络与窄通道高维数网络相比,其延迟较低、冲突较少、热点吞吐量较大。

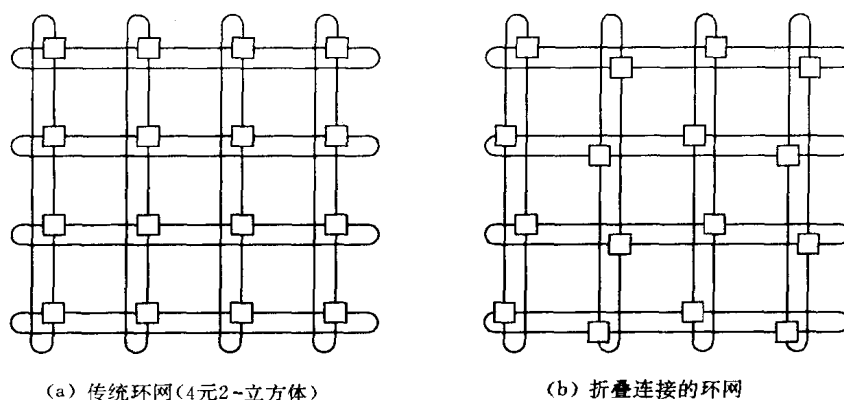


图 7.18 环网中线长相等的折叠连接

表 7.1 汇总了静态互连网络的重要特性。大多数网络的结点度都小于 4, 这是比较理想的。全连接网络和星形网络的结点度都太高。超立方体的结点度随 $\log_2 N$ 值增大而增大, 当 N 值很大时其结点度也太高。

表 7.1 静态网络特性一览表

网络类型	结点度 d	网络直径 D	链路数 l	等分带宽 B	对称性	网络规格评注
线性阵列	2	$N - 1$	$N - 1$	1	非	N 个结点
环形	2	$\lceil N/2 \rceil$	N	2	是	N 个结点
全连接	$N - 1$	1	$N(N - 1)/2$	$(N/2)^2$	是	N 个结点
二叉树	3	$2(h - 1)$	$N - 1$	1	非	树高 $h = \lceil \log_2 N \rceil$
星形	$N - 1$	2	$N - 1$	$\lceil N/2 \rceil$	非	N 个结点
2D 网格	4	$2(r - 1)$	$2N - 2r$	r	非	$r \times r$ 网络, $r = \sqrt{N}$
Illiac 网	4	$r - 1$	$2N$	$2r$	非	与 $r = \sqrt{N}$ 的带弦环等效
2D 环网	4	$2\lceil r/2 \rceil$	$2N$	$2r$	是	$r \times r$ 环网, $r = \sqrt{N}$
超立方体	n	n	$nN/2$	$N/2$	是	N 个结点, $n = \log_2 N$ (维数)
CCC	3	$2k - 1 + \lceil k/2 \rceil$	$3N/2$	$N/(2k)$	是	$N = k \times 2^k$ 结点, 环长 $k \geq 3$
k 元 n -立方体	$2n$	$n\lceil k/2 \rceil$	nN	$2k^{n-1}$	是	$N = k^n$ 个结点

网络直径的变化范围很大。但随着硬件寻径技术不断革新(虫蚀寻径), 直径已不是一个严重问题, 因为任意两结点间的通信延迟在高度流水线操作下几乎是固定不变的。链路数会影响网络价格, 等分宽度将影响网络的带宽。

对称性会影响可扩展性和寻径效率。客观地说, 网络的总价格随 d 和 l 增大而上升。直径小仍然是一种优点。但是, 结点间的平均距离可能是一种更好的量度指标。等分宽度可以用较宽的通道宽度来扩大。根据以上分析, 环、网格、环网 k 元 n -立方体和 CCC 都具备

一定的条件用以建造未来的 MPP 系统。

2. 动态互连网络

动态互连网络设置有源开关,因而可根据需要借助控制信号对连接通路加以重新组合,实现所要求的通信模式。下面我们介绍总线、多级互连网路和交叉开关网络。

(1) **总线** 总线系统实际上是一组导线和插座用于处理与总线相连的处理机、存储模块和外围设备间的数据业务。总线只用于源(主部件)和目的(从部件)之间一回处理一次业务。在多个请求情况下,总线仲裁逻辑必须每次能将总线服务分配或重新分配给一个请求。

基于这一原因,总线被称为多个功能模块间的争用总线(contention bus)或时分总线(time-sharing bus)。总线系统与其他两种动态连接网络相比,其价格较低,带宽较窄。它有很多可用的工业和 IEEE 总线标准。

图 7.19 所示的是一种总线连接的多处理机系统。系统总线在处理机或 I/O 子系统和存储模块或辅助存储设备(磁盘、磁带机等)之间提供了一条公用通信通路。系统总线通常设置在印刷电路板底板上。其他的处理器板、存储器板或设备接口板都通过插座或电缆插入底板。

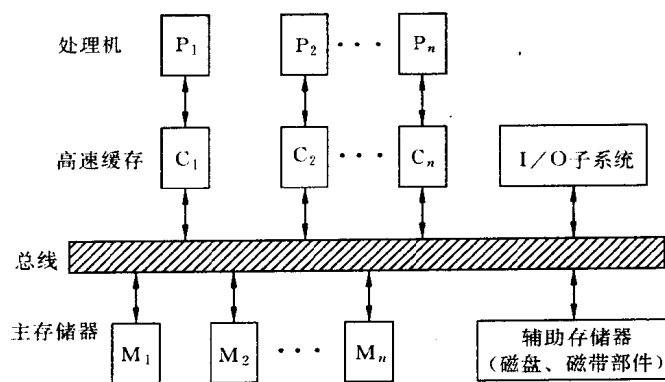


图 7.19 一种总线连接的多处理机系统

主动设备或主设备(处理机或 I/O 子系统)产生访问存储器的请求,被动设备或从设备(存储器或外围设备)则响应请求。公用总线是在分时基础上工作的,而总线研制中的重要问题有总线仲裁、中断处理、一致性协议和总线事务的处理等。

(2) **开关模块** 一个 $a \times b$ 开关模块有 a 个输入和 b 个输出。一个二元开关则与 $a = b = 2$ 的 2×2 开关模块相对应。在理论上, a 和 b 不一定要相等,然而实际上 a 和 b 经常选为 2 的整数幂,即 $a = b = 2^k, k \geq 1$ 。

表 7.2 列出了几种常用的开关模块大小: 2×2 , 4×4 和 8×8 。每个输入可与一个或多个输出相连,但是在输出端必须避免发生冲突。换句话说,一对一和一对多映射是容许的;但不容许有多对一映射,因为输出端将发生冲突。

在只容许一对一映射(置换)时,我们称这种模块为 $n \times n$ 交叉开关。例如, 2×2 交叉开关可有两种连接模式:直送和交叉。一般说来,一个 $n \times n$ 交叉开关可实现 $n!$ 置换。在

表 7.2 中还列出了不同大小开关模块的合法连接模式的数目。

表 7.2 开关模块和合法状态

模块大小	合法状态	置换连接
2×2	4	2
4×4	256	24
8×8	16 777 216	40 320
$n \times n$	n^n	$n!$

(3) **多级网络** MIMD 和 SIMD 计算机都使用多级网络。一种通用多级网络如图 7.20 所示,其中每一级都用了多个 $a \times b$ 开关,相邻各级开关之间都有固定的级间连接。为了在输入和输出之间建立所需的连接,可用动态设置开关的状态来实现。

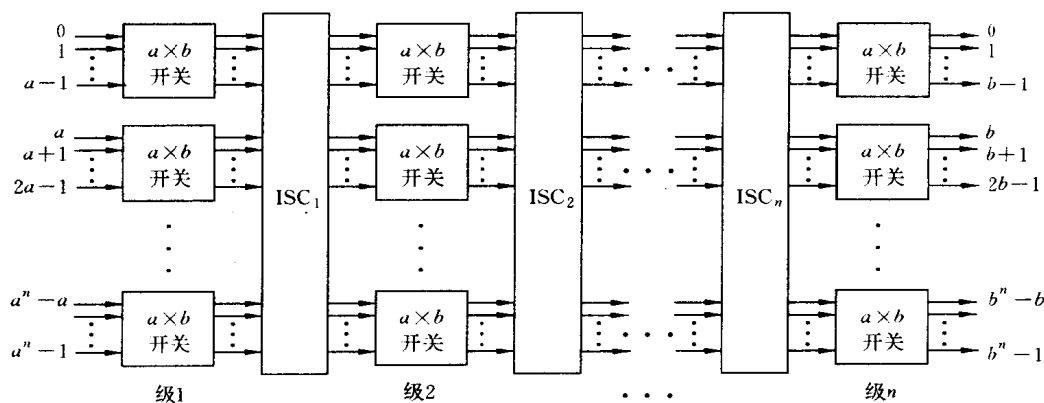


图 7.20 一种由 $a \times b$ 开关模块和级间连接模式 $ISC_1, ISC_2, \dots, ISC_n$ 构成的通用多级互连网络结构

各种多级网络的区别就在于所用开关模块、控制方式和级间连接(ISC)模式的不同。最简单的开关模块是 2×2 开关。控制方式是对各个开关模块进行控制的方式,它可以有 3 种:级控制——每一级的所有开关只用一个控制信号控制,同时只能处于同一种状态;单元控制——每一个开关都有自己单独的控制信号控制,可各自处于不同的状态;部分级控制——第 i 级的所有开关分别用 $i+1$ 个信号控制, $0 \leq i \leq n-1, n$ 为级数。常用的 ISC 模式包括有均匀洗牌、蝶式、多路洗牌、纵横交叉、立方体连接等。

(4) **Ω 网络** 图 7.21(a)至 7.21(d)画出了用于构造 Ω 网络的 2×2 开关四种可能的连接方式。图 7.21(e)所示的是一个 $16 \times 16 \Omega$ 网络,共需 4 级 2×2 开关。网络左侧有 16 个输入,右侧有 16 个输出。ISC 是对 16 个对象的均匀洗牌模式。

一般说来,一个 n 输入的 Ω 网络需要 $\log_2 n$ 级 2×2 开关,每级要用 $n/2$ 个开关模块,网络共需 $n \log_2 n/2$ 个开关。每个开关模块采用单元控制方式。

不同的开关状态组合可实现各种置换、广播或从输入到输出的其他连接。

(5) **基准网络** Wu 和 Feng 研究过多级互连网络之间的关系。基准网络可如图 7.22 (a)所示递归生成。

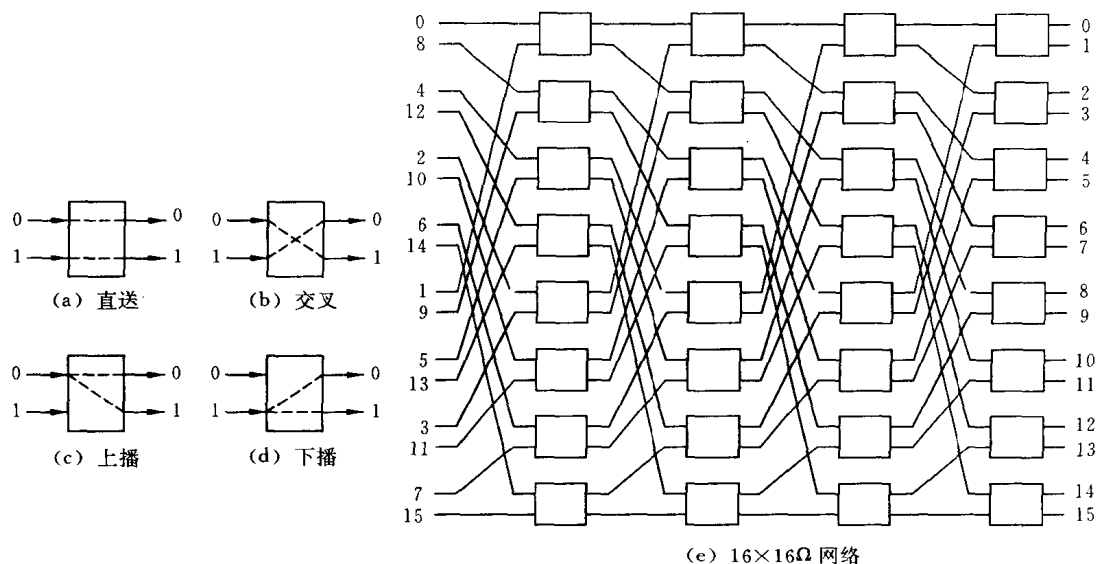


图 7.21 用 2×2 开关和均匀洗牌作为级间连接模式构成一个 16×16 Ω 网络

第一级为一个 $N \times N$ 模块,第二级为两个 $(N/2) \times (N/2)$ 子模块,以 C_0 和 C_1 表示。以上构成方法可递归用于子模块,直至得到 2×2 的 $N/2$ 子模块为止。各小框和最终的子模块构件是 2×2 开关,每个有两个合法连接状态:两个输入和两个输出间的直送和交叉连接。 16×16 基准网络如图 7.22 所示。

(6) **交叉开关网络** 交叉开关网络的带宽和互连特性最好。它可看作是一个单级开关网络。像电话交换机一样,交叉点开关能在对偶(源、目的)之间形成动态连接,每个交叉点开关在对偶间提供一条专用连接通路,开关可根据程序的要求动态地设置“开”或“关”。图 7.23 所示的是两种交叉开关网络。

我们可以在处理机和存储模块之间用交叉开关网络构成一个共享存储型多处理机(图 7.23),实际上这是一个存储器访问网络。 $C.mmp$ 多处理机已实现了 16×16 交叉开关网络,它将 16 台 PDP-11 处理机与 16 个存储模块连在一起,每个存储模块的容量为 1M 存储单元字。16 台处理机最多可同时访问 16 个存储模块。

注意,每个存储模块一次只能满足一台处理机的请求。在多个请求同时到达同一存储模块时,交叉开关就必须分解所发生的冲突,每个交叉开关的行为与总线非常相似。但是,每台处理机可能会产生一系列地址要同时访问多个存储模块。因此,在图 7.23(a)中每一列只能接通一个交叉点开关。但是,为了支持并行(或交叉)存储器访问,可以同时接通几个交叉点开关。

还有一种交叉开关网络可用于处理机间通信,如图 7.23(b)所示。Fujitsu 公司(1992)制造的向量并行处理机(VPP 500)实际上就是采用了这种大型交叉开关网络(224×224)。其中 PE 为接有存储器的处理机,CP 代表控制处理机,用来监控整个系统,包括交叉开关网络的运行。在这种网络中,每一行和每一列只能接通一个交叉点开关。

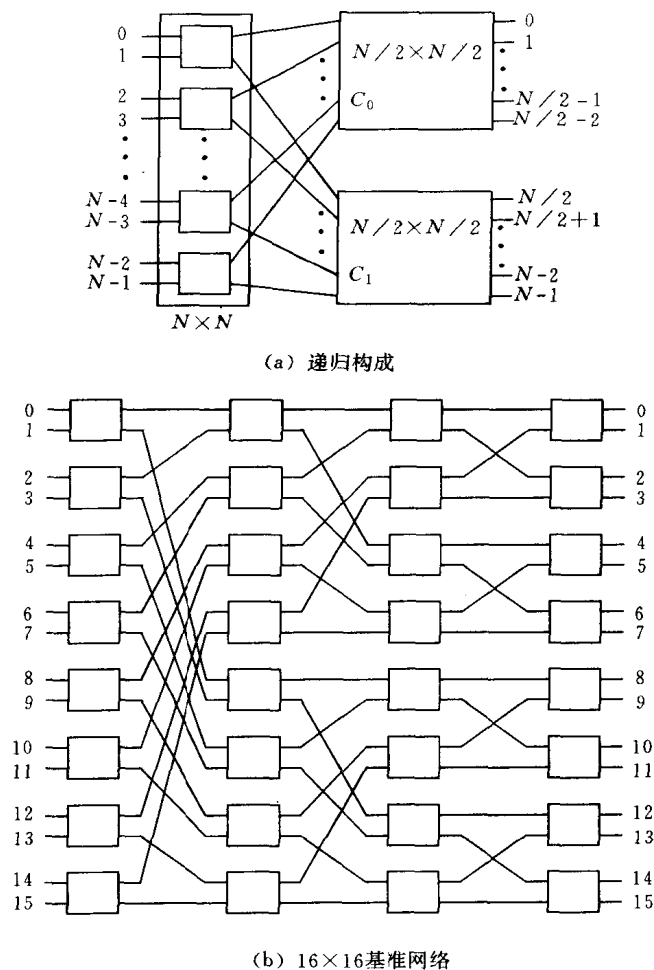


图 7.22 基准网络的递归构成

处理机间的交叉开关可实现处理机之间的置换连接,但这只是一对一的连接。所以 $n \times n$ 交叉开关网络一次最多可连通 n 个(源,目的)对,这是与处理机存储器间的交叉开关网络不同的地方。

在表 7.3 中,我们汇总了构成动态网络的总线、多级网络、交叉开关的主要特性。显然,总线的造价最低,但其缺点是每台处理机可用的带宽较窄。

总线所存在的另一个问题是容易产生故障。有些容错系统,如用于事务处理的 Tandem 多处理机等,常采用双总线以防止系统产生简单的故障。由于交叉开关的硬件复杂性以 n^2 上升,所以其造价最为昂贵。但是,交叉开关的带宽和寻径性能最好。如网络的规模较小,它是一种理想的选择。

多级网络则是两个极端之间的折衷。它的主要优点在于采用模块结构,因而可扩展性较好。然而,其时延随网络的级数 $\log n$ 而上升。另外,由于增加了连线和开关复杂性,价格也是一种限制因素。

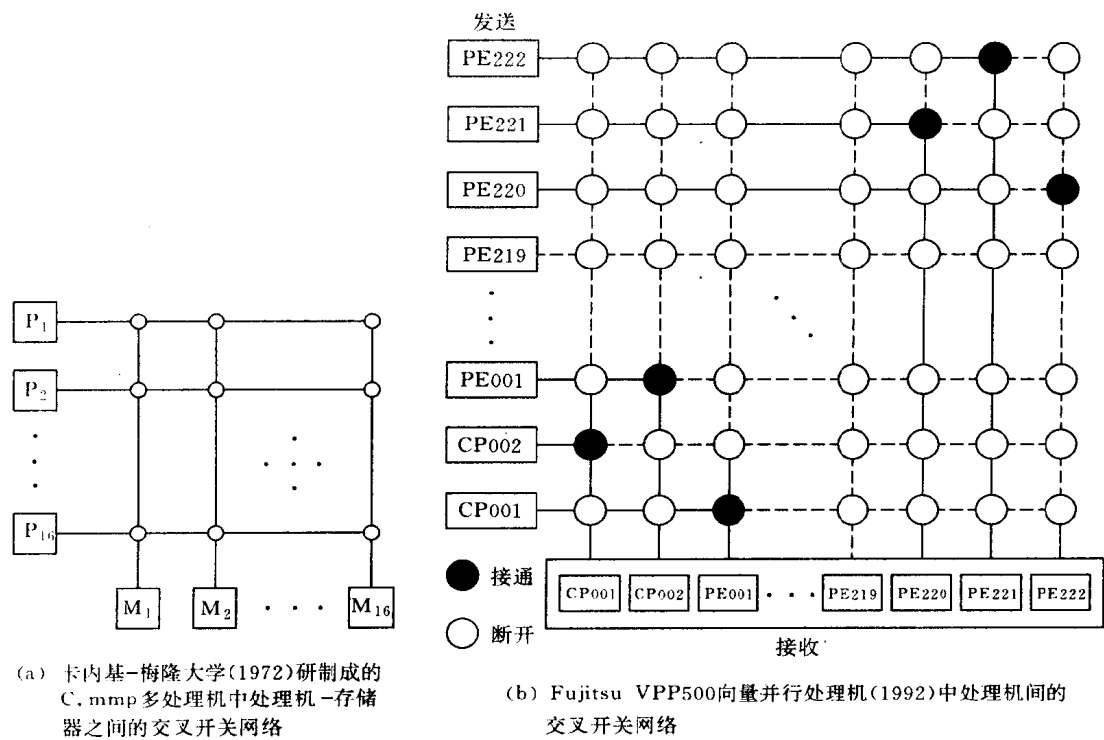


图 7.23 两种交叉开关网络构型

表 7.3 动态网络特性一览表

网络特性	总线系统	多级网络	交叉开关
单位数据传送的最小时延	恒定	$O(\log_k n)$	恒定
每台处理机的带宽	$O(w/n)$ 至 $O(w)$	$O(w)$ 至 $O(nw)$	$O(w)$ 至 $O(nw)$
连线复杂性	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
开关复杂性	$O(n)$	$O(n \log_k n)$	$O(n^2)$
连接特性和寻径性能	一次只能一对一	只要网络不阻塞,可实现某些置换和广播	全置换,一次一个
典型计算机	Symmetry S1, Encore Multimax	BBNTC-2000 IBM RP3	Cray Y-MP/816 Fujitsu VPP 500
评注	总线上假定有 n 台处理机;总线宽度为 w 位	$n \times n$ MIN 采用 $k \times k$ 开关,其线宽为 w 位	假定 $n \times n$ 交叉开关的线宽为 w 位

互连网络也可以如图 7.24 那样分类。

(1) **共享介质网络** 这种网络包含总线形和环形两类,它们在同一时间都只允许一个设备进行存取。其中总线包括底板总线、争用总线(以太网 Ethernet)和令牌总线(ARC-net)。底板总线又包括单总线(SGI POWERpath-2, DEC LSB)和双总线(SUN XDBus)。主要的环形有 FDDI 环和 IBM 令牌环。

(2) **非阻塞网络** 这种网络可以被认为是逻辑的交叉开关网络。除非存在不同的输

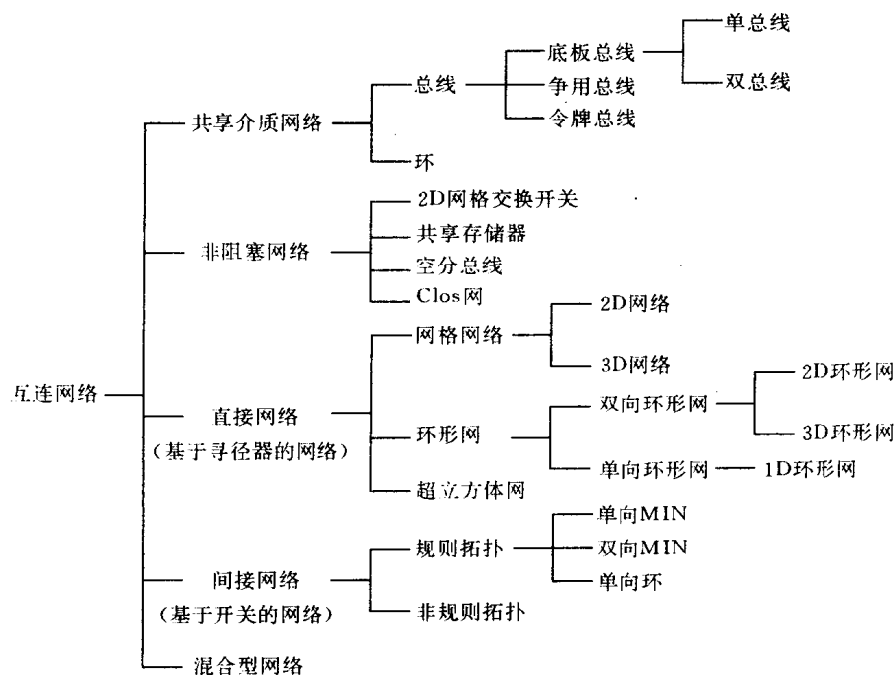


图 7.24 互连网络分类

入端口同时向同一个输出端口发送消息，否则消息通信将不会阻塞。有许多方法可以用来设计无阻塞网络，从物理的交叉开关设计到空分的总线设计。由于交叉开关的硬件复杂性以 N 的平方上升，所以其造价最为昂贵，但是，交叉开关的带宽和寻径性能最好，又由于设计的空间复杂性，因此无阻塞网络主要用于构筑小规模的高性能的互连网络，一般从 4 个端口到 32 端口。这些小规模的无阻塞网络还可以用作构筑更大规模的无阻塞网络的基本部件。

无阻塞网络包括 2 维网格的交换开关网络 (Cray X/Y-MP, DEC GIGAswitch, Myrinet)、共享存储器网络 (CNET Prelude)、空分总线 (Fore ATMASX-100) 和 Clos 网络。

(3) **直接网络** 是指网络中的处理器是点到点连接的，也被称作静态网络，或称作基于寻径器的网络，因为所有的结点都有一个寻径器用来处理结点间的消息通信。一般地，相邻的两结点通过一对相反方向的单向通道连接。也可以用一个双向的通道连接，如果是采用这种方案的话，必须有一个仲裁协议来决定使用通道的哪一侧。每一个寻径器都有一些内部的输入、输出通道与本地的处理器相连接，还有一些外部的输入、输出通道与相邻的结点相连。正是这些外部的通道决定了互连网络的拓扑结构。网格、环形网和超立方体是典型的直接网络的拓扑结构。

直接网络可以分为三类：网格、环形网和超立方体。其中网格分为 2 维网格 (Intel Paragon) 和 3 维网络 (MIT J machine)。环形网分为双向的环形网和单向的环形网，其中双向的环形网有 2 维的 (Intel/CMU iWARP) 和 3 维的 (Cray T3D)。KSR 的第一层环采用单向的一维环形网。采用超立方体的典型的机器有 Intel iPSC 和 n CUBE。

(4) **间接网络** 也称为基于开关的网络,是另一类适于大规模互连网络的网络拓扑结构。这类网络中结点不是通过直接相连的通道进行消息通信,而是通过网络的开关机构进行的。每一个结点有一个网络适配器连接到网络的开关上。这些互连方式决定了网络的拓扑。大多数网络采用的是各种规则的多级互连网络的拓扑结构。

间接网络可以分为规则拓扑和不规则拓扑,其中规则的拓扑结构又可以分为单向的多级网络(NEC Cenju-3)、双向的多级网络(IBM SP,TMC CM-5,Meiko CS-2)和单向的环(KSR-2-level ring, Conver Exemplar)。

(5) **混合网络** 指一个互连网络中混合了多种以上的网络。例如,一个互连网络有一个超立方体的主干网络,同时每个结点是一个网格网络。在 Convex Exemplar 中,每个结点是 8 个处理器的交叉开关网络,而这些结点用 4 个重复的单向环相连。

7.2 消息传递机制

消息传递机制的研究在互连网络的研究中占有非常重要的地位。在多计算机系统中通过互连网络进行消息传递需要专门的硬件和软件的支持。这一节将研究各种寻径方法,并分析它们的通信时延问题。我们还要引入虚拟通道的概念,考察消息传递网络产生死锁的各种情况,说明怎样用虚拟通道来避免死锁。

为了实现无死锁的消息寻径,人们提出了确定的和自适应两种寻径算法。我们先来讨论确定的维序寻径方法,例如用于超立方体的 E 立方体寻径和用于二维网格的 X-Y 寻径即属于这一类方法。然后再讨论虚拟通道或虚拟子网的自适应算法。

需要说明的是,拓扑结构的选择不是与寻径策略毫无关系。例如,拓扑结构在很大程度上决定了各类不同寻径方案的可用性及是否可以采用自适应寻径。上面两个问题影响着一些关键的网络特性,如可容错性和可模块化等,因此我们对不同的拓扑结构考虑的寻径策略是不一样的。

7.2.1 消息寻径方式

消息格式的改进使两代多计算机的寻径由存储转发方式演进为虫蚀方式,下面首先介绍消息的格式。接着再讨论在通信路径上相继的寻径器之间的异步流水握手协议。最后还要对时延进行分析,说明现有的寻径方式在时间上的差别。

1. 消息格式

消息寻径中的信息单位如图 7.25 所示。消息是结点间通信的逻辑单位,它常常由任意数目的长度固定的包组成,因此它的长度是可变的。

包是包含寻径目的地址的基本单位。由于不同的包可能异步地到达目的结点,因此每个包需要一个序号,以便把传递的消息重新装配起来。

可以进一步把包分成一些固定长度的数据片。寻径信息(目的地址)和序号形成头片,其余的片是数据片。

在采用存储转发寻径方式的多计算机系统中,包是信息传送的最小单位。在采用虫蚀

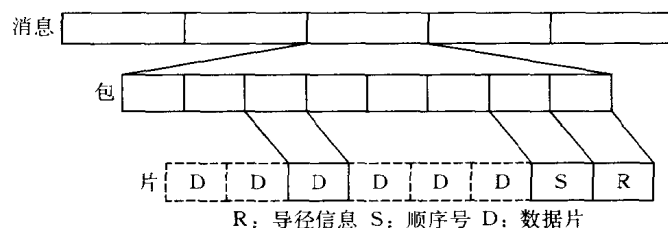


图 7.25 在消息传递网络中通信的信息单位:消息、包和片的格式

寻径网络的多计算机中,包可以进一步分成片。片的长度往往受网络大小的影响。256 个结点的网络需要片长为 8 位。

包的长度取决于寻径方式和网络的实现方法。典型的包的长度为 64 位~512 位。序号可能占用 1~2 个片,取决于消息的长度。包和片的大小还与通道频宽、寻径器设计以及网络流量密度等有关。

2. 四种寻径方式

消息寻径方式可以分为两大类:线路交换和包交换。其中包交换又包括:存储转发寻径、虚拟直通寻径和虫蚀寻径等。下面逐一进行讨论。

(1) 线路交换(circuit switch) 在线路交换这种寻径方式下,在传递一个消息之前,先建立一条从源结点到目的结点的物理通路,然后再传递消息。如图 7.26(a)所示。

其传输时延用公式表示为 $T = (L_h/B) \times D + L/B$, L_h 为建立路径所需的小信息包的长度, L 为信息包的长度, D 为经过的结点数, B 为带宽(以下同)。

在并行计算机中的频繁的小信息包通信的这种方式下,由于在传递一个消息之前,需要频繁地建立从源结点到目的结点的物理通路,开销将会很大,与以下的几种包交换(packet switch)的寻径方式相比这是一个很大的缺点。包交换的寻径方式以其较高的传输带宽和较低的平均传输时延,更适于具有动态和突发特性的 MPP 数据传送。

(2) 存储转发寻径(store and forward) 在存储转发网络中包是信息流的基本单位。每个结点有一个包缓冲区。包从源结点经过一系列中间结点到达目的结点。

当一个包到达一个中间结点时,它首先被存入缓冲区。当所要求的输出通道和接收结点的包缓冲区可使用时,然后再将它传送给下一个结点。如图 7.26(b)所示。

存储转发网络的时延与源和目的地之间的距离(段数)成正比。第一代多计算机系统采用这种寻径方式。其时延用公式表示为 $T = (L/B) \times D + L/B = (D + 1) \times L/B$ 。

可以看到,存储转发寻径有两个很大的缺点:

- 包缓冲区大,不利于 VLSI 的实现
- 时延大,与结点距离成正比

(3) 虚拟直通(virtual cut through) 目前有一些多计算机系统采用的是虚拟直通的寻径方式。虚拟直通的寻径方式的思想是,为了减少时延,没有必要等到整个消息全部缓冲后再作路由选择,只要接收到用作寻径的消息头部即可判断。

其通信时延用公式表示为 $T = (L_h/B) \times D + L/B = (L_h \times D + L)/B$, L_h 是消息的寻径头部的长度。一般来说, $L \gg L_h \times D$, 所以公式可以近似为 $T = L/B$, 可以看到此时通信时延与结点数无关,这相对于存储转发的寻径方式来说是一个非常大的改进。

然而,当出现寻径阻塞时,虚拟直通方式只有将整个消息全部存储在寻径结点中,直到寻径通道不阻塞时才能将消息发出,这就需要每个寻径结点都有足够的缓冲区来存储可能出现的最大的信息包,在这一点上,虚拟直通方式与存储转发的寻径方式是一样的,同样不利于 VLSI 的实现。因此,虚拟直通方式在最坏的情况下与存储转发方式的通信时延是一样的。由此出现了下面将要讨论的新的寻径方式——虫蚀寻径方式——它改进了以上提到的缺点。

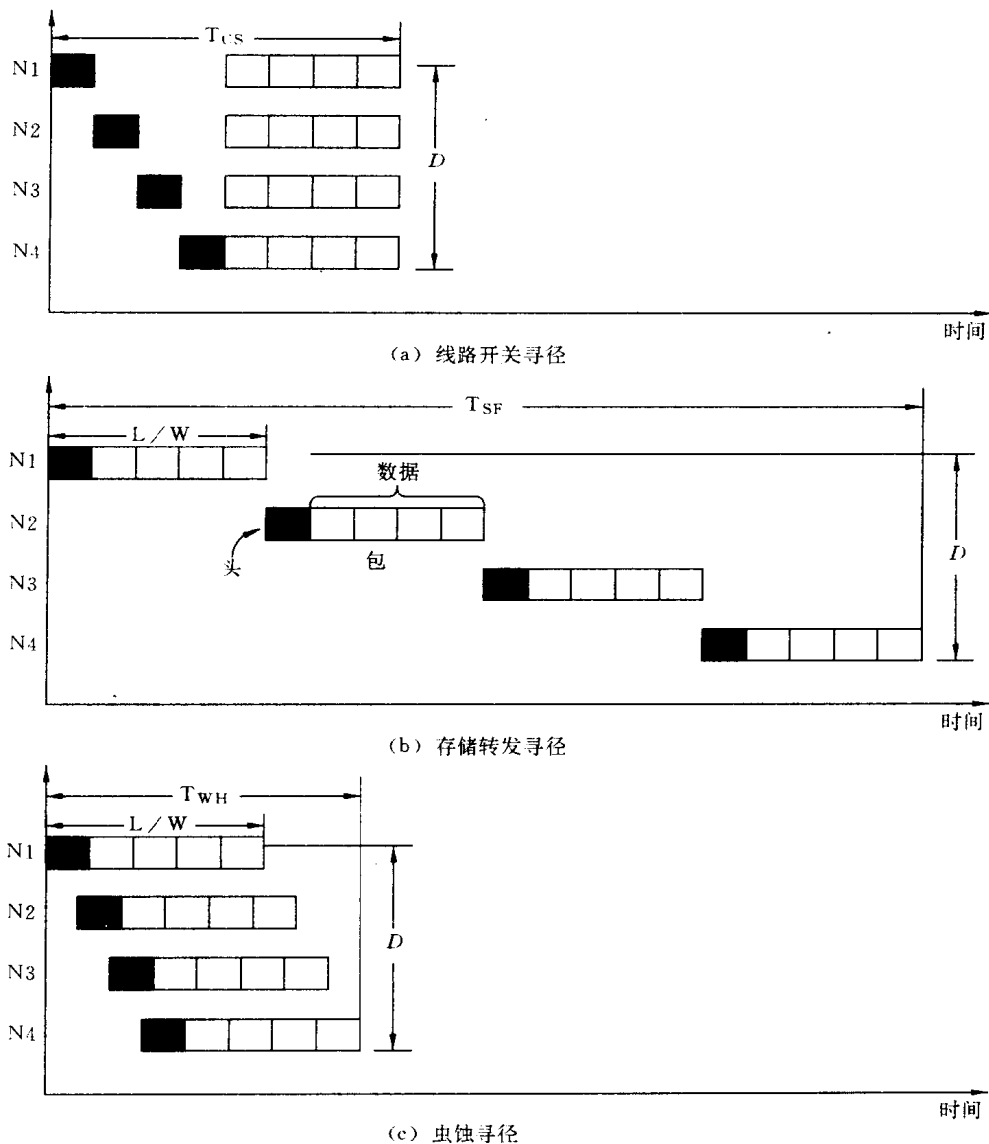


图 7.26 几种寻径方式的时空图

(4) 虫蚀寻径(wormhole) 新型的多计算机系统很多采用的是虫蚀寻径方式,把包进一步分成更小的片。与结点相连的硬件寻径器中有片缓冲区。消息从源结点传送到目

的结点要经过一系列寻径器。如图 7.26(c)所示。

同一个包中所有的片像不可分离的同伴一样以流水方式顺序地传送。包可以看作是一列火车,由火车头(头片)和被牵引的车厢(数据片)组成。

只有头片知道包将发往何处。所有的数据片必须跟着头片。不同的包可以交替地传送,但不同包的片不能交叉,否则它们可能被送到错误的目的地。

用头片直接开辟一条从输入链路到输出链路的路径的方法来进行操作。每个消息中的片以流水方式在网络中向前“蠕动”。每个片相当于虫的一个节,“蠕动”是以节为单位顺序地向前爬行。

当消息的头片到达一个结点 A 的寻径器后,寻径器根据头片的寻径消息立即做出路由选择:

如果所选择的通道空闲且所选择的结点 B 的片缓冲区可用,那么这个头片就不必等待,直接通过结点 A 传向下一个结点 B。随后的其他数据片跟着相应地向前“蠕动”一步。当消息的尾片向前“蠕动”一步之后,它刚才所占有的结点就被放弃了。

如果所选择的通道忙或所选择的结点的片缓冲区不可用时,那么这个头片就必须在该结点的片缓冲区中等待,直到上述两者都可用时为止,其他数据片也在原来的结点上等待。此时,被阻塞的消息不从网络中移去,片也不放弃它所占有的结点和通道。

为了实现上述的一个包内相继片的异步流水操作,采用如图 7.27 所示的握手协议。沿着路径,相邻寻径器之间有一根一位的就绪/请求(R/A)线。当接收寻径器(D)就绪时(图 7.27(a)),就可以接收一片,即片缓冲区可用了,R/A 线的电平变低。当发送的寻径器(S)就绪时(图 7.27(b)),R/A 线的电平变高并通过通道传送片 i 。当 D 正在接收片时(图 7.27(c)),R/A 线保持高电平。当片 i 从 D 的缓冲区移走后(即传送到下一个结点)(图 7.27(d)),重复上述操作过程以便传送片 $i+1$,直到整个包都被接收。

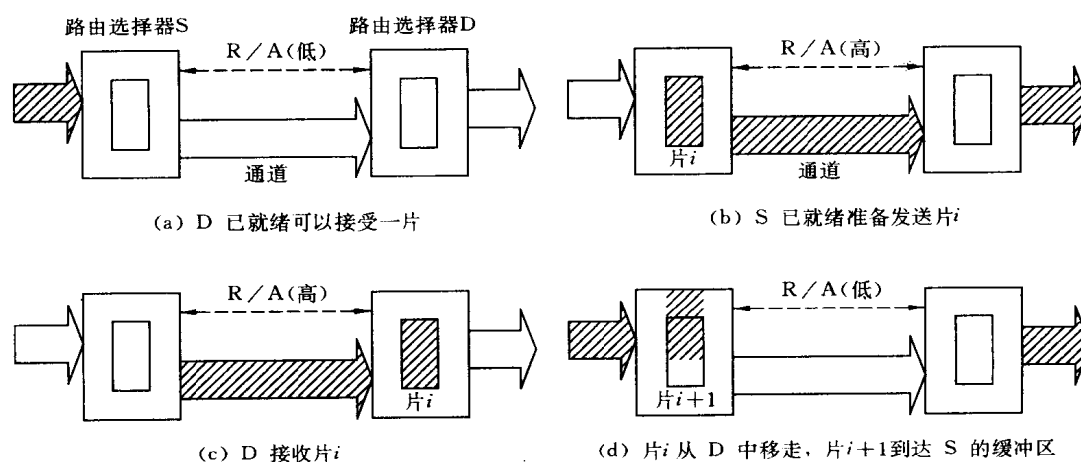


图 7.27 两个虫蚀寻径器之间的握手协议

虫蚀寻径的通信时延用公式表示为 $T = T_f \times D + L/B = (L_f/B) \times D + L/B = (L_f \times D + L)/B$, L_f 是片的长度, T_f 是片经过一个结点所需时间。一般来说, $L \gg L_f \times D$, 所

以公式可以近似为 $T = L/B$ ，可以看到此时通信时延也与结点数无关。

可以看出，虫蚀寻径有以下的优点：

- 每个结点的缓冲区较小，易于 VLSI 实现。
- 较低的网络传输时延。所有的片以流水方式向前传输，采用了时间并行性。而存储转发方式的消息包整个地从一个结点“跳”到另一个结点，通道的使用是串行的，所以它的传输时延基本上正比于消息在网络中传输的距离。虫蚀寻径方式的网络传输时延正比于消息包的长度，传输距离对它的影响很小。
- 通道共享性好，利用率高，对通道的预约和释放是结合在一起的一个完整的过程，有一段新的通道后将立即放弃用过的一段旧通道。
- 易于实现选播和广播通信方式。允许寻径器复制消息包的片并把它们从其多个输出通道输出。

然而虫蚀寻径方式也有缺点。当消息的一个片被阻塞时，整个消息的所有片都将被阻塞在所在结点，占用了结点资源，因此需要采用虚拟通道的方式来避免由此引起的一连串的阻塞。

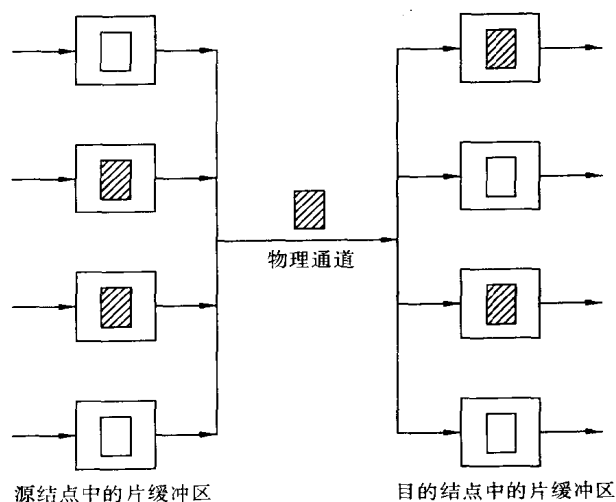
虫蚀寻径方式也可以分为无缓冲和有缓冲两类，区别在于缓冲的大小。缓冲大有利于性能的提高，但会增加结点的复杂度。IBM SP2 采用的寻径方式就是带缓冲的虫蚀寻径方式，它采用共享的存储区来对输入/输出消息进行缓冲。图 7.26 是线路开关寻径、存储转发寻径和虫蚀寻径的时空图。

7.2.2 死锁和虚拟通道

虫蚀寻径多计算机网络的通信通道实际上由许多源和目的对共享。从共享物理通道可以引出虚拟通道的概念。这一节我们将介绍这一概念并讨论它在避免死锁方面的应用。

1. 虚拟通道

虚拟通道是两个结点间的逻辑链，它由源结点的片缓冲区、结点间的物理通道以及接收结点的片缓冲区组成。图 7.28 说明了四条虚拟通道共享一条物理通道的概念。



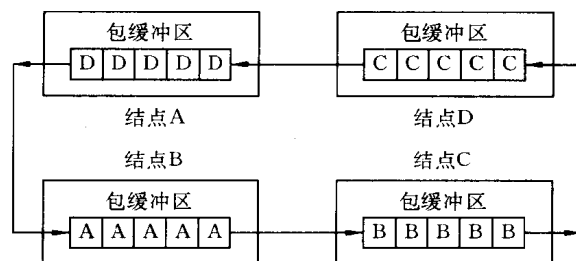
源结点和接收结点各有 4 个片缓冲区。当物理通道分配给某对缓冲区时,这一对的源缓冲区和接收缓冲区形成了一条虚拟通道。

换句话说,物理通道由所有的虚拟通道分时地共享。除了有关的缓冲区和通道之外,还必须用某些通道状态(如 R/A 信号)来表示不同的虚拟通道。源缓冲区存放等待使用通道的片。接收缓冲区存放由通道刚刚传送过来的片。通道(电缆或光纤)是它们之间的通信媒介。

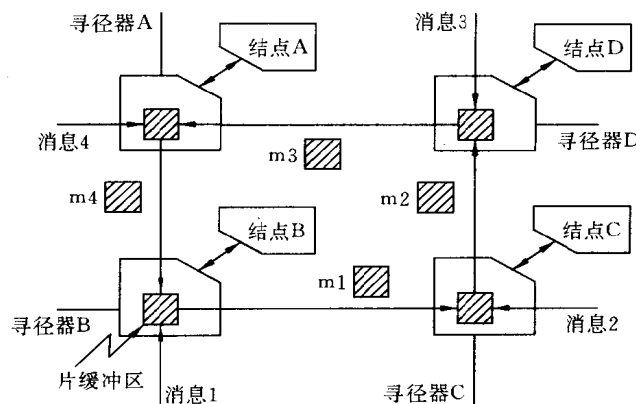
2. 死锁的产生和避免

缓冲区或通道上的循环等待会引起死锁,如图 7.29 所示。

图 7.29(a)是存储转发网络中缓冲区死锁。四个消息包占用了四个结点的四个缓冲区,导致循环等待。除非扬弃某个消息包或者某个包的寻径出错,否则死锁不会解除。在图 7.29(b)采用虫蚀寻径的网格形网络中,四个消息沿四条通道同时传送也会产生通道死锁。四个消息的四个片同时占用了四条通道。如果循环中没有一条通道被释放,死锁状态将持续下去。



(a) 采用存储转发寻径的四个结点之间出现缓冲区死锁



(b) 采用虫蚀寻径的四个结点之间出现通道死锁
(带阴影方块是片缓冲区)

图 7.29 缓冲区或通道上的循环等待引起的死锁

那么如何避免死锁呢? 图 7.30 是通道相关图。图中,结点表示通道,带方向的箭头表示通道之间的依赖关系。利用虚拟通道方法可以避免死锁。通过增加两条虚拟通道 V_3 和 V_4 ,如图 7.30(c)所示,可以打破死锁循环,增加虚拟通道 V_3 和 V_4 可得到一张修改后的

通道相关图,在使用通道 C_2 之后不再使用 C_3 和 C_4 。

将图 7.30(b)中的环路转变成螺旋线就可以避免死锁。通道多路复用可在片一级进行,如果包长度足够的话,那么也可以在包一级进行。

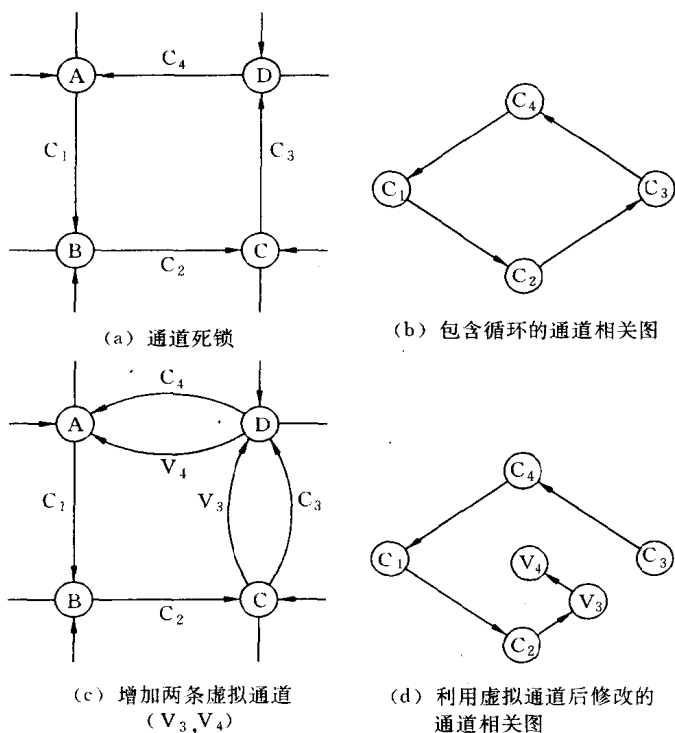


图 7.30 利用虚拟通道避免死锁

虚拟通道可以用单向通道或者双向通道实现。把两条单向通道组合在一起可以构成一条双向通道,这不仅增加了利用率而且还可使通道的频宽加倍。

然而,双向通道中的仲裁要复杂一点。用双向通道互连的相邻结点需要专用的仲裁线,用它来控制信息流的方向。

实际上,和单向通道相比较,双向通道由于要做方向仲裁,因而增加了延迟,又由于控制复杂,因而还增加了成本。如果网络的流量不大,则双向通道的效率比较高。

虚拟通道可能会使每个请求可用的有效通道频宽降低。确定虚拟通道数目时,需要对网络吞吐量和通信时延折衷考虑。实现数目很大的虚拟通道需要用高速的多路选择开关。

7.2.3 流控制策略

当两个或更多的包在某个结点为竞争缓冲区或通道资源而发生冲突时,必须确定如何解决冲突的策略。这一节将考察各种不会引起拥挤或死锁现象的控制网络流量的策略。下面将以这些策略为基础,讨论为一对一通信设计的确定寻径算法和自适应寻径算法。

1. 包冲突的解决

通道流水线上的两个相邻结点之间要传送片时,必须具备三个条件:(1) 源缓冲区已存有该片;(2) 通道已分配好;(3) 接收缓冲区准备接收该片。

当两个包到达同一个结点时,它们可能会请求用同一个接收缓冲区或者要用同一个输出通道。因此必须对两个问题做出仲裁:(1)把通道分配给哪个包?(2)没有分配到通道的包做什么事?结果有四种解决包冲突的方法,如图 7.31 所示。

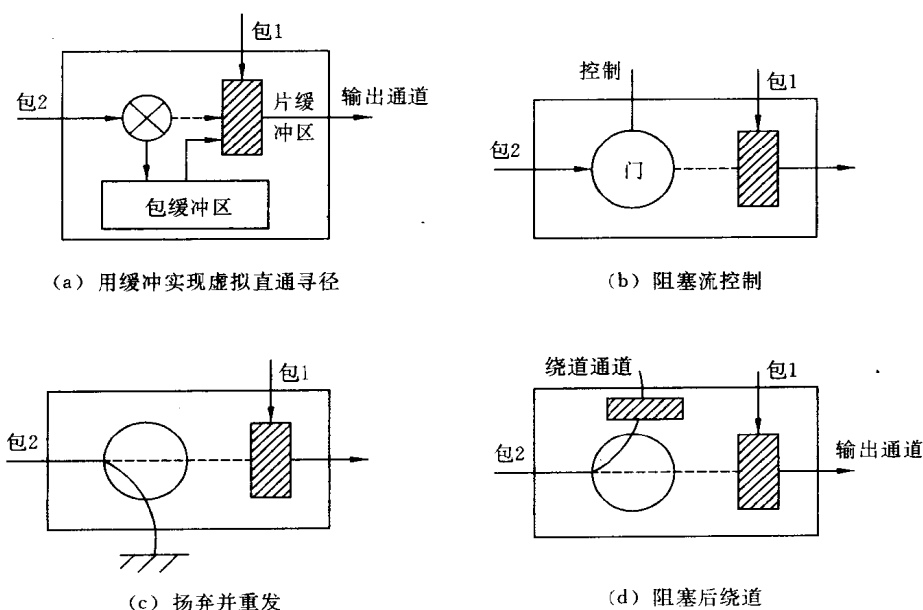


图 7.31 解决两个包请求同一条输出通道发生冲突时的流控制方法

图 7.31 说明当两个包争用某个中间结点的同一条输出通道时,有四种解决这一冲突的方法。把通道分配给包 1,拒绝包 2。Kermani 和 Kleinrock(1979)提出了一种虚拟直通寻径(virtual cut-through routing)缓冲方法。包 2 被暂时存放在一个包缓冲区。当通道可以使用时再传送包 2。这种缓冲方法的好处是不会浪费已经分配的资源;但是需要一个能存放整个包的缓冲区。此外,通信路径上的包缓冲区不应形成如图 7.29(a)所示的循环。由于包缓冲区不可能做在寻径器芯片上,因此要用本地存储器作为包缓冲区,这会引起相当大的存储延迟。虚拟直通方法是存储转发和虫蚀两种寻径方法的折衷。当不发生冲突时,就如同虫蚀寻径方法一样工作。在最坏情况下,效果则与存储转发寻径方法相同。

纯粹的虫蚀寻径在出现冲突时就采用阻塞(blocking)策略,如图 7.31(b)所示。第 2 个包被阻塞不再前进,但是并没有被场弃。图 731(c)所示的是扬弃(discard)策略,它简单地把被阻塞的包扬弃掉。

第四种策略称为绕道(detour)(见图 7.31(d))。被阻塞的包被送到一条绕行的通道。阻塞策略的实现成本低,但可能出现分配给阻塞包的资源空闲的情况。扬弃策略可能会出现资源严重浪费,并且需要包重新发送和回答,否则被扬弃的包也许会丢失。现在已很少使用这种策略,因为包的输送率不稳定。BBN Butterfly 网络采用了这种扬弃策略。绕道寻径为包寻径提供了更大的灵活性。然而,绕道可能要用更多的通道资源才能到达目的地。而且绕行的包可能进入一个活锁(livelock)循环,这将浪费网络资源。Connection Machine 和 Denelcor HEP 采用了这种绕道策略。实际上,某些多计算机网络综合了以上某些流控制策略的优点,采用混合策略。

2. 确定寻径和自适应寻径

下面讨论如何找出一条从源结点到目的结点的路径来传送消息。寻径可以分为确定和自适应两类。采用确定寻径时,通信路径完全由源和目的地址确定。换句话说,寻找的路径是预先唯一确定的,与网络状况无关。自适应寻径与网络状况有关,可能会有几条路径。这两种寻径都需要无死锁算法。

(1) 确定寻径

下面给出两种基于维序概念的确定寻径算法。

维序寻径是一种按照多维网络维序的特定顺序来选择后继通道。在二维网格网络中称为 X-Y 寻径,首先沿着 X 维方向确定路径,然后沿着 Y 维方向选择路径。在超立体(或 n 立方体)网络中,采用最初由 Sullivan 和 Bashkow(1977)提出的称为 E 立方体寻径(E-cube routing)方法。

- 二维网格网络的 X-Y 寻径 假定从任意源结点 $s = (x_1, y_1)$ 到任意目的结点 $d = (x_2, y_2)$, 寻径从 s 开始,首先沿 X 方向前进一直到 d 所在的第 x_2 列为止,然后沿 Y 方向前进直到 d 。

与东-北、东-南、西-北及西-南的路径方向相对应,X-Y 寻径共有四种模式。图 7.32 是二维网格连接多计算机的 X-Y 寻径的例子。图中有 4 个(源,目的)对,可用以说明二维网格的四种寻径模式。

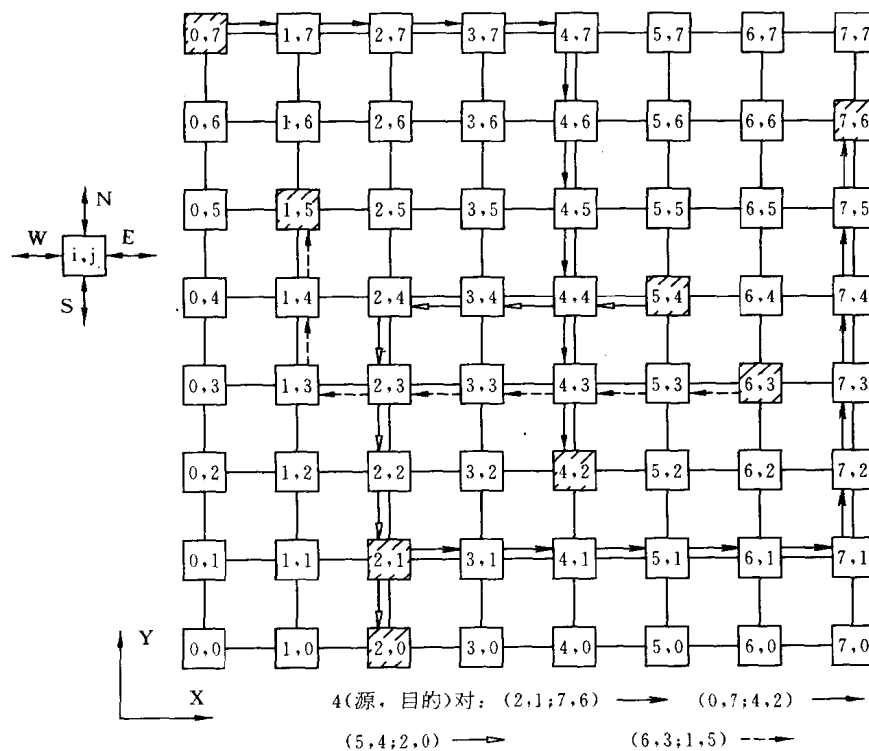


图 7.32 64(即 8×8)个结点二维网格计算机的 X-Y 寻径

从结点(2,1)到结点(7,6)需要一条东-北路径。从结点(0,7)到结点(4,2)要建立一条

东-南路径。从结点(5,4)到结点(2,0)需要一条西-南路径。从结点(6,3)到结点(1,5)需要一条西-北方向路径。如果总是先沿 X 维方向寻径,然后再沿 Y 维方向寻径,寻径就不会出现死锁或循环等待现象。

按照维序,可以很容易地将 X-Y 寻径扩充到 n 维网络。以三维网格为例,可以类似地说明 X-Y-Z 寻径方法。X-Y 方式不会产生死锁寻径,可用于存储转发或虫蚀寻径网络,在源和目的结点之间形成一条距离最短的路径,然而对于环网网络,采用维序寻径方法不能得到最短路径。为了减少网络流量或其他原因,不会产生死锁的非最短寻径算法有时会使包通过的路径较长。

• 超立方体网络的 E 立方体寻径 假设有一 $N = 2^n$ 个结点的 n 方体。每个结点的二进制编码为 $b = b_{n-1}b_{n-2}\cdots b_1b_0$ 。这样,源结点为 $s = s_{n-1}\cdots s_1s_0$,目的结点 $d = d_{n-1}d_{n-2}\cdots d_1d_0$ 。现在要确定一条从 s 到 d 的步数最小的路径。

将 n 维表示成 $i = 1, 2, \dots, n$, 其中第 i 维对应于结点地址中的第 $i-1$ 位。设 $\nu = \nu_{n-1}\cdots\nu_1\nu_0$ 是路径中的任一结点。路径可以根据以下方法唯一地确定。

① 计算方向位 $r_i = s_{i-1} \oplus d_{i-1}$, 其中 $i = 1, 2, \dots, n$ 。

使 $i = 1, \nu = s$, 开始以下步骤。

② 如果 $r_i = 1$, 则从当前结点 ν 寻径到下一结点 $\nu \oplus 2^{i-1}$ 。如果 $r_i = 0$, 则跳过这一步。

③ $i \leftarrow i + 1$ 。如果 $i \leq n$, 则转第 2 步, 否则退出。

下面用图 7.33 中的例子来说明上述 E 方体寻径算法。例中, $n = 4, s = 0110, d = 1101$, 因此 $r = r_4r_3r_2r_1 = 1011$ 。由于 $r_1 = 0 \oplus 1 = 1$, 因此 s 就寻径到 $s \oplus 2^0 = 0111$ 。由于 $r_2 = 1 \oplus 0 = 1$, 因此 $\nu = 0111$ 就寻径到 $\nu \oplus 2^1 = 0101$ 。由于 $r_3 = 1 \oplus 1 = 0$, 因此就可跳过维 $i = 3$ 这一步。由于 $r_4 = 1$, 因此 $\nu = 0101$ 就寻径到 $\nu \oplus 2^3 = 1101 = d$ 。

所选择的路径在图 7.33 中用箭头所示。注意, 寻径是按照从维 1 到维 4 的顺序进行的。如果 s 和 d 的第 i 位相同, 则沿维 i 方向不需要寻径, 否则从当前结点沿着这一维方向走到其他结点, 重复这一过程直到到达目的结点。

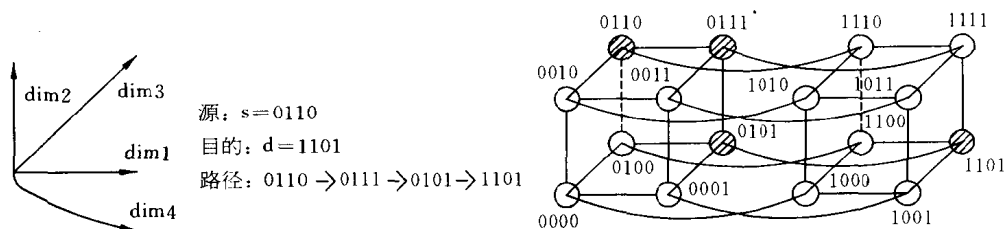


图 7.33 16 个结点超立方体计算机的 E 方体寻径

E 立方体寻径也不会产生死锁寻径,也可用于存储转发和虫蚀网络,在源和结点之间形成一条距离最短的路径。

(2) 自适应寻径

采用自适应寻径要特别注意避免死锁。虚拟通道的概念使实现自适应寻径更经济 and 更灵活。图 7.34 说明怎样利用虚拟通道达到这一目的。在网络连接网络中,同一维的所有连接都使用虚拟通道。

图 7.34 是一个用 X-Y 寻径的网格网络,在 Y 维上用了 2 对虚拟通道。图 7.34(c)中的虚拟网络可以用来避免消息在向西传输出现的死锁,因为所有向东的 X 通道都没有使用。同样,图 7.34(d)中的虚拟网络使用另一组 X 方向虚拟通道来支持只向东的传输。在不同的时刻使用两个虚拟网络,这样死锁就可以自动避免。

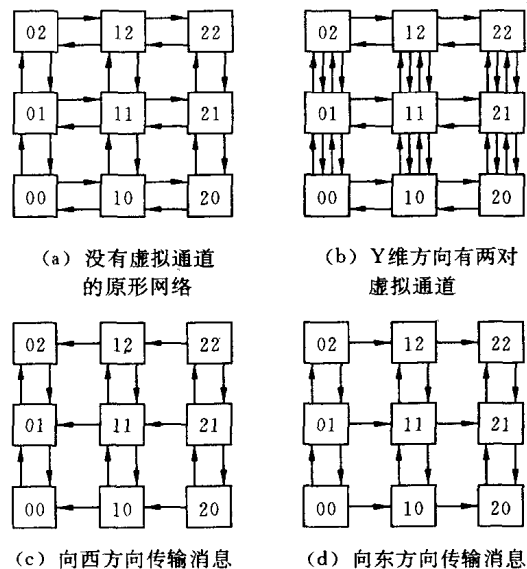


图 7.34 利用虚拟通道避免死锁的自适应 X-Y 寻径

图 7.35 是在 X 维和 Y 维方向各有两条虚拟通道的网格形网络。这些虚拟通道可以用来生成 4 个虚拟网络。西-北方向通信应该使用图 7.35(b)所示的虚拟网络。类似地,其他方向的通信可以构造另外三个虚拟网络。注意,任何一个虚拟网络都不会出现环路。因此,在这些网络上实现 X-Y 寻径方法时,完全可以避免死锁。

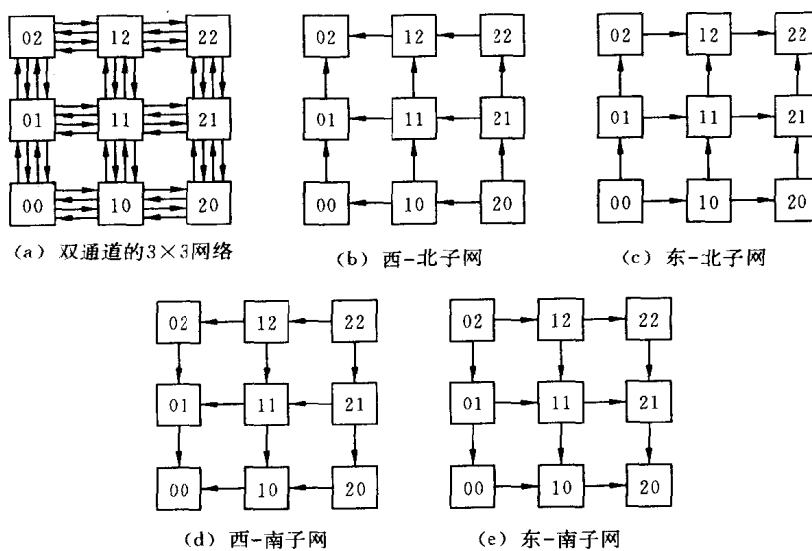


图 7.35 双通道网格网络可实现四个虚拟网络

如果相邻结点之间的两对通道都是物理通道,那么4个虚拟网络中任何两个都可以同时使用而不会产生冲突。如果相邻结点之间双虚拟通道只能共享一对物理通道,那么只有(b)和(c)或(c)和(d)可以同时使用。其他组合如(b)和(c),或(b)和(d),或(c)和(e),或(d)和(e)都不能同时存在,因为缺少通道。

显然,网络的通道数增加,寻径的自适应性也将增加。然而,成本的增加将很可观,因此要避免使用过多的资源。

7.2.4 选播和广播寻径算法

多计算机网络中会出现四种通信模式:

(1) 单播(unicast)模式对应于一对一的通信情况,即一个源结点发送消息到一个目的结点。

(2) 选播(multicast)模式对应于一到多的通信情况,即一个源结点发送同一个消息到多个目的结点。

(3) 广播(broadcast)模式对应于一到全部的通信情况,即一个源结点发送同一个消息到全部结点。

(4) 会议(conference)模式对应于多到多的通信情况。

前一节讨论的是单播模式,下面来分析一下实现选播、广播或会议通信模式的要求。当然,所有的各种模式都可以顺序地多次使用单播来实现,或者如果不会发生源冲突的话,甚至也可以同时用多个单播来实现。实现这些多目的模式必须使用特殊的硬件或寻径方法。

通道流量(channel traffic)和通信时延(communication latency)是描述效率常用的两个参数。通道流量可用传输有关消息所使用的通道数来表示。通信时延则用包的最长传输时间来表示。

优化的寻径网络应以最小流量和最小时延实现有关通信模式。然而,这两个参数并不是毫不相关的。达到最小流量时不一定能达到最小时延,相反的情况也是如此。

这与所使用的交换技术有关,在存储转发网络中时延是最重要的问题,而在虫蚀寻径网络中流量对效率的影响则更大。

下面讨论网络连接计算机中的选播和广播。

图7.36是在 3×4 网格上实现的选播寻径。源结点用S表示,传送一个包到标号 D_i 的5个目的结点。这里, $i=1,2,\dots,5$ 。目的结点为5个的选播可以用5次单播来实现,如图7.36(a)所示。X-Y寻径的流量需要用 $1+3+4+3+2=13$ 条通道,到 D_3 的路径最长,所以时延是4。

选播还可以用下述方法实现,即在一个中间结点上复制所传送的包,然后把该包的多个拷贝送到目的结点,这样可以大大减少通道流量。

图7.36(b)和7.36(c)给出了两种选播寻径模式,流量分别为7和6。在虫蚀寻径网络中,用图7.36(c)的选播寻径模式比较好。在存储转发网络中,则用图7.36(b)的寻径模式比较好,而且时延较短。

使用一棵4层的生成树可以把一个包从结点S广播到所有的网格结点,如图7.36(d)所示。到达树第 i 层上结点的时延为 i 。这种广播树产生的时延和流量都最小。

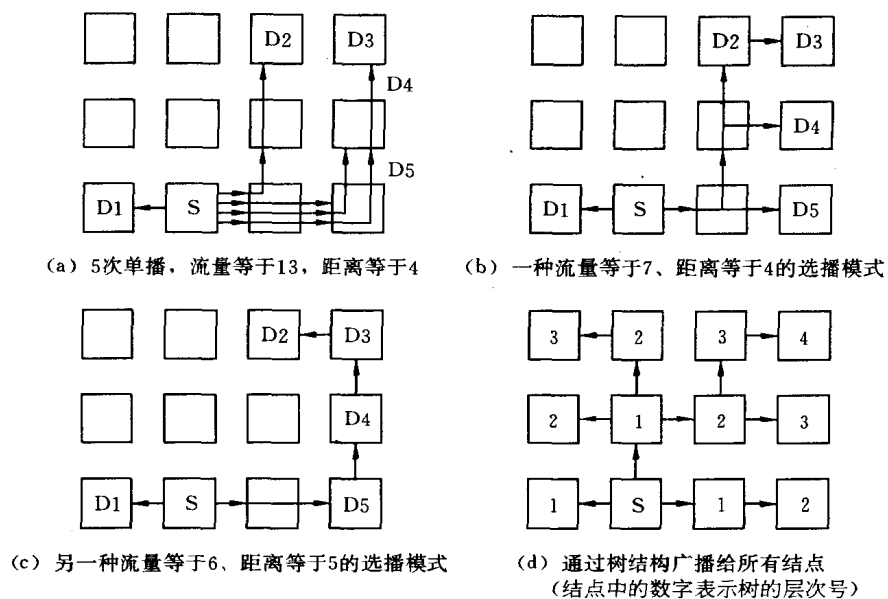


图 7.36 3×4 网格计算机上的多次单播、选播模式和广播树

下面讨论超立方体计算机上的选播和广播。

为在 n 方体上实现广播, 可用类似的生成树, 时延不超过 n 就能到达所有结点。图 7.37(a) 是一个根结点为 0000 的 4 立方体。超立方体广播树的流量最小。

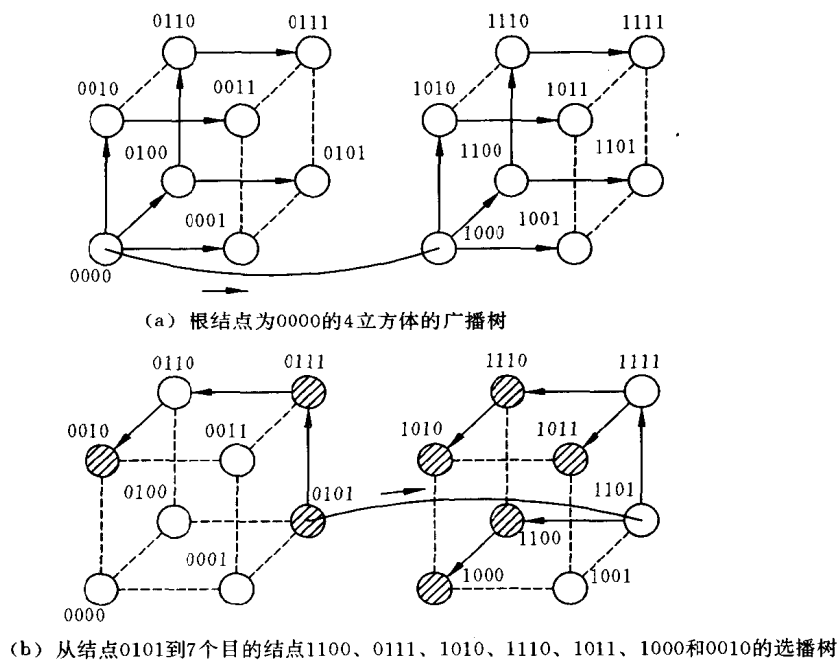


图 7.37 采用贪婪算法 4 立方体的广播树和选播树

图 7.37(b) 是一棵贪婪选播树, 可从结点 0101 发送包到 7 个目的结点。这个贪婪选播算法的基本思想是向那些可达到最多剩余目的结点的维方向发送包。

从源结点 $S=0101$ 开始, 由维 2 方向可以到达 2 个目的结点, 由维 4 方向可以达到 5 个目的结点。因此, 第一层所用的通道是 $0101 \rightarrow 0111$ 和 $0101 \rightarrow 1101$ 。

从结点 1101, 由维 2 方向可以到达 3 个目的结点, 由维 1 方向可以到达 4 个目的结点。因此, 第二层所用的通道是 $1101 \rightarrow 1111$, $1101 \rightarrow 1100$ 和 $0111 \rightarrow 0110$ 。

同理, 第三层所用的通道是 $1111 \rightarrow 1110$, $1111 \rightarrow 1011$, $1100 \rightarrow 1000$ 和 $0110 \rightarrow 0010$ 。第四层所用的通道是 $1110 \rightarrow 1010$ 。

在扩充选播树时, 首先应该比较所有各维方向的可达性(reachability), 然后选择某些维使剩余目的结点的集合最小。如果两维之间有连线, 那么选择其中任何一维都可以。因此, 所生成的树不是唯一的。

已经证明贪婪选播算法所需的通道数与多次单播或广播树相比要少。在虫蚀寻径网络中, 实现选播操作时, 每个结点的寻径器应有复制片缓冲区中数据的能力。

为了与选播树或广播树的生长同步, 树中同一层的所有输出通道必须在传输向前推进一层之前处于就绪状态, 否则中间结点需要增加缓冲区。

7.3 互连网络实例

这一节将着重研究下面几种多处理机的互连网络

1. 总线结构
2. 多级开关网络
3. 超立方体
4. 交叉开关

以上并没有包括所有多处理机的互连网络, 但却是最典型和最常见的互连网络。下面将从低成本低带宽的互连网络到低成本高带宽的互连网络一一讨论, 所要研究的主要问题是一致的。

我们可以得到这样的结论: 多处理机系统采用哪种互连结构主要取决于系统的最大通信量。反过来, 系统的最大通信量受到互连结构的限制。如果系统工作在通信饱和点以下, 互连结构可以通过改变 R/C 比值来调节系统的性能。一个好的设计应该尽量使系统工作在通信饱和点以下, 这样可以通过提高 R/C 比值来获得高的吞吐率。 R/C 比值问题将在第九章中讨论。

7.3.1 总线互连

在讨论系统性能时, 我们特别强调效率, 并说明了 R/C 比值的重要作用。一种能满足效率要求, 构成多处理机的最简单方法是通过共享总线把各台处理机连结起来, 再配备各处理机都可访问的全局存储器。图 7.38 是总线互连的多处理机框图。

每台处理机都能访问公共总线。全局存储器与公共总线相连。全局存储器是为所有处理机共享的全局资源。每台处理机还拥有各自的局部存储器和 Cache 存储器。这样可

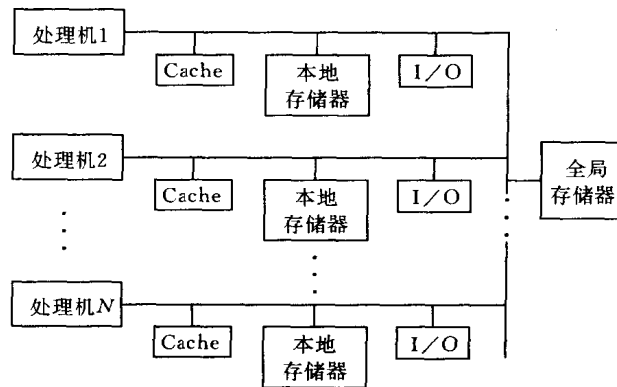


图 7.38 总线连结的多处理机

以降低各处理机通过共享总线访问全局存储器的频度,从而减少由于争用而造成对系统性能的不良影响。

如果没有局部存储器和 Cache 存储器,那么访存的开销将变得非常高,这是因为在这种情况下各台处理机都要频繁地访问全局存储器,必然会出现严重的总线争用现象,使得选择总线的延迟过长,大大降低多处理机系统的性能。远程访问和竞争造成的很长延迟会明显增大 R/C 中的 C 值,这就减小了加速比以及进行计算用的处理机数目。使用局部存储器和 Cache 存储器的目的就在于缩短访存周期,降低处理机使用总线的频度,以保证每台处理机不会由于总线干扰而降低其他处理机的工作效率。如果局部存储器和 Cache 存储器可以使处理机使用总线的次数减少 90%,那么在相同的总线争用条件下,就能使互连处理机总数比没有局部存储器和 Cache 存储器的系统增加 10 倍。如果能减少 95% 的总线使用次数,互连处理机总数可以增加 20 倍。

我们期望面向总线的系统能够支持 10 台处理机有效地工作,可能的话支持 20 台,30 台处理机。如果超过这一范围,总线争用将使系统性能严重下降。从这个意义上说,总线互连结构显然不能支持 1 000 台以上的处理机并行工作,除非器件技术出现惊人的突破——生产出带宽极高而成本很低的总线。其实,即使真的发生这种技术突破,也只能使系统瓶颈从总线转移到共享存储器。共享存储器是除总线之外另一个最易发生争用的资源,而且它很容易达到饱和状态。也就是说,即使总线带宽可以支持 1 000 台处理机,但共享存储器最多只能满足 100 台处理机的要求。

在面向总线的系统结构中使用 Cache 存储器将会出现许多具体问题。这些问题是由于要保持所有 Cache 存储器中数据的一致性才产生的。如果某个 Cache 存储器的某一数据被修改了,那么当另一台处理机访问相应 Cache 存储器的相应数据时,读到的应该被修改过的新数据。这就要求 Cache 存储器的控制必须遵守如下协议——无论数据在本地 Cache 存储器或在远程 Cache 存储器,还是在共享存储器中,都要保证每台处理机存取的数据是正确的值。

为了实现这种协议——保证 Cache 存储器的数据一致性,往往必须增加一些额外的共享总线操作。如果不采用 Cache 存储器,这些附加操作都可以忽略。因此,采用 Cache 存

储器的系统结构当 Cache 存储器的命中率较高时,可以减少对共享总线的使用。但是,一致性协议而引起的额外总线处理在一定程度上又抑制了 Cache 存储器的作用。

为了使面向总线的多处理机具有实用性,采用什么样的技术手段是非常重要的。含有 N 台处理机的多处理机系统所需的总线带宽应该是单处理机带宽的 N 倍,所以总线带宽限制了处理机的台数。如果总线及其接口采用非凡技术,而处理机内部采用普通技术,那么采用非凡技术所增加的成本并不会太高,而随着互连处理机台数的增加,多处理机的性能将大大提高。因此,采用速度要比处理机所采用的技术快 100 倍,并能支持 1000 台处理机的总线互连是合理的。

如果通信线路采用非凡技术,那么同样的技术也可以应用到整个多机系统中,以使每台处理机的吞吐率有近百倍的提高。这样的话,很可能由 10 台采用非凡技术的处理机组成的系统,就可以承担由 1000 台普通处理机组成的系统所能完成的任务,当然其总线也必须采用非凡技术。

总线是面向总线型多处理机的一个“瓶颈”,共享存储器是系统的另一个“瓶颈”。随着通信带宽的增大,系统性能最终将受到共享存储器带宽的限制。这是因为进程通过读写共享存储器单元以达到同步工作,随着进程个数的增加,某些共享单元被访问的次数也将增加。

例如,我们来看一个控制 N 台处理机执行的称作阻塞器的存储单元。进程运行到阻塞器时便被挂起,处于等待状态直到所有需同步的进程都到达为止,然后继续执行。阻塞单元的值最初为 N , 当一个进程到达时,其值就减 1,直到阻塞单元的内容等于零,所有进程被释放。

如果共享单元每次只能被一台处理机访问,那么使阻塞器从 N 变为 0 所需的时间为 $O(N)$ 。如果并行运行的进程正在执行某项需要常数时间的工作,那么当 N 值足够大时,阻塞单元本身将成为系统瓶颈,它会在很大程度上限制系统的工作效率。

为了克服共享存储器这一瓶颈,我们应该从技术、系统结构和算法等几方面去寻找解决方法。

1. 技术方面:使用高速器件组成共享存储器,或者采用非凡的存储器技术以支持多路并行存储访问。
2. 系统结构方面:设计具有高带宽的系统,以支持共享和控制。
3. 算法方面:对于专门的应用问题,寻找分布式控制方法以减小或消除集中式控制变量所造成的瓶颈。

以上各种方法都是行得通的,采用任意一种方法都可能建立一个符合设计要求的系统。然而,遗憾的是我们并不能保证上述设计方法都能取得成功。

现在研究应该采用什么样的技术来实现总线。高速总线的长度必须非常短,这个限制非常重要。因为高速意味着电流和电压变化非常快,而这些物理量的变化速度受到电容和电感的限制。因为寄生电容和电感随导线长度正比例地增加,所以必须缩短连线的物理距离。信号经过较长距离的传递后,其传真度就会下降,这会增大传输过程中出错的概率。因此,如果总线较长或存在其他减慢传输速度和降低传真度的因素,则其带宽一定低于信号质量较高、长度较短的总线带宽。干扰噪声则是另一个问题,它是由于近邻信号互相干扰

造成的,这种干扰同样也随总线的物理距离增大而增加。

我们面临的问题是随着总线所连接的处理机数目的增加,大多数总线的带宽将下降。因此,不仅每台处理机都要与其余 $N - 1$ 台处理机共享总线带宽,而且随着 N 的增大可用的共享带宽也要下降。适于较小 N 值的总线技术对于较大 N 值的情况就不一定能适用。随着 N 的增大,总线效率将从有效逐渐变为无效。其确切的分界点则取决于所采用的技术,应该根据总线的具体类型及接口技术来判断。

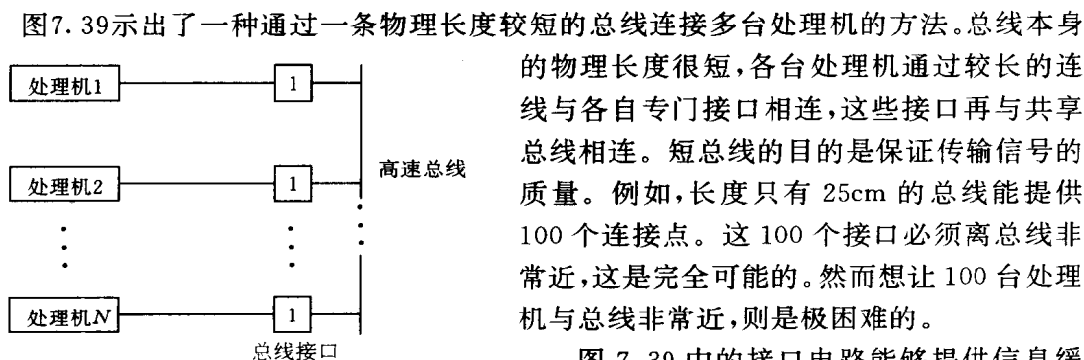


图 7.39 高速总线结构

总线本身的物理长度很短,各台处理机通过较长的连线与各自专门接口相连,这些接口再与共享总线相连。短总线的目的是保证传输信号的质量。例如,长度只有 25cm 的总线能提供 100 个连接点。这 100 个接口必须离总线非常近,这是完全可能的。然而想让 100 台处理机与总线非常近,则是极困难的。

图 7.39 中的接口电路能够提供信息缓冲,这样允许处理机之间有一定的距离以满足处理机芯片封装的要求。虽然图 7.39 中的

总线是在处理机模块的外面,但图示的结构对某些使用超级 VLSI 技术的系统也是适用的。这时,总线以及各台处理机被装配在同一芯片中。

7.3.2 环形互连

虽然总线互连对于处理机台数较少的多机系统来说有一定的优越性,但是总线性能将受到物理因素的限制。系统结构设计者的目标是要寻找一种互连方法,它既能具有总线型互连的简单性,又可以克服总线所固有的缺点。一种可能的方法是构造一种逻辑总线。环形互连就属于这种结构。

图 7.40 是处理机之间点点连接的环形互连网络。在此系统中,信息的传送过程是发送进程把信息放到环上,这些信息便通过环形网络不断向下一台处理机传播,直到此信息回到发送者为止。

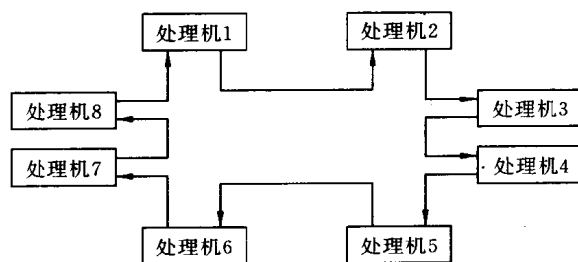


图 7.40 基于环形连接的多处理机结构

这种环形网络的操作方法很多,例如 IEEE 802.5 令牌环(token ring)标准是把环形

看作逻辑总线的协议。发送信息的处理机拥有一个唯一的令牌,在同一时刻只有一台处理机持有这个令牌。当发送者发送信息时,这个环形网络的作用如同总线一样,其他处理机都处于接收信息的状态。

信息传送结束时,发送者播送一个令牌,这个令牌是在普通信息中不会出现的特定信号。每台处理机依次看到令牌。如果某台处理机等待发送信息,那么它便接收这个令牌而不再传递给下一台处理机,这时这台处理机就可以通过环形网络发送信息了。假如没有想发送信息的处理机,那么令牌就将在环形网络上不断循环,直到某台处理机需要发送信息为止。

令牌环的优点是点点连接,而不是总线连接,其物理参数更容易控制。事实上,令牌环形互连非常适于高带宽的光纤通信。 N 较小的总线互连系统很难采用光纤通信, N 较大的系统也还未实现。

令牌环形互连的一个主要缺点是每个总线接口都有延迟,通常是 1 位的延迟。当互连处理机台数增加时,在环中的信息传输延迟将正比例地增加。但是当系统负载很重时,系统的带宽却不会像总线互连系统那样下降。

为了发挥令牌环形互连的优点,系统结构设计者可以把令牌环形看成是一种周期短延迟长的流水线。只要计算时能够保持流水线处于充满状态,其有效带宽就可以被充分利用。因此各台处理机应该把计算和传输重叠进行。

此外,高速环形互连网络还应遵守如下协议:发送者发送完全部信息之后,不等回收到此信息就应把其持有的令牌传递给新的发送者。这种协议保证了信息在环形网络上以流水方式传递,这对于充分利用有效带宽是必要的。如果只有当原来的信息都不在环上传递时才能发送新的信息,那么网络的效率就如同在两次不同操作之间都要排空流水线一样。这样,随着环形网络上处理机台数的增加,必然会造成严重的带宽下降。

采用目前的技术可以使长度较短的总线的工作频率达到 10~50 MHz,具体是多少则和总线长度及最大负载有关。显然,总线长度越长,负载越重,工作速度就越低。如果总线能封装在一片 VLSI 芯片里,那么总线的速度将非常快,甚至能超过 100 MHz。如是总线不能封装在一个芯片里,那么其最高时钟频率会降到 10~50 MHz。这时,只能精心安排提高封装密度,并尽可能地减小寄生电容和电感,才能加快速度。采用光连接的令牌环形网络的工作速度要快得多。

7.3.3 交叉开关互连

总线互连最简单,但争用最严重。交叉开关互连则是不同于总线互连的另一种方法。这种方法把争用现象降到最低程度,但连接复杂度最高。在这一节讨论交叉开关互连,下一节讨论介于总线和交叉开关之间的其他一些互连方法。

图 7.41 是交叉开关网络,它把 N 台处理机和 N 个存储器连接起来。图中处理机台数与存储器个数相等,但是一般情况是存储器数目等于或数倍于处理机数目。网络中每个交叉点是一个允许任何一台处理器与任何一个存储器连接的开关。

任一处理机与任一存储器进行信息交换时只在交叉开关处有一个单位的延迟。通信网络中没有争用现象。假如处理机 P_1 要访问存储器 M_1 ,同时处理机 P_2 要访问存储器

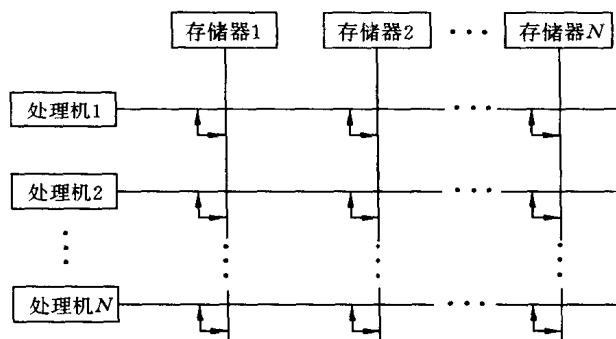


图 7.41 基于交叉开关连接的多处理机

M_2 , 那么它们可以并行操作。实际上, 系统允许 N 个存取操作并行执行, 当然任意两个存取操作不能涉及同一台处理机或同一个存储器。

如果两个或两个以上存取操作同时要访问某个存储器, 那么争用现象就发生了。例如处理机 P_1 和 P_2 同时要访问存储器 M_1 , 显然其中有一台处理机要等待另一台处理机完成访问后才能进行访问。

为了减少争用, 在系统结构方面大有文章可做。例如, 如果争用是由于多台处理机同时要访问同一存储模块中不同单元的数据而引起的, 一种解决争用的方法是尽量使这些数据平均地分配到多个不同的存储模块中, 而不要把它们集中在一个存储模块中。例如把一个数据块中数据依次分配到相继的存储模块中。同样, 共享程序代码也应该这样分配。这样, 当两台或两台以上的处理机同时访问共享程序代码或数据时, 争用只使其中一台处理机延迟一个周期。只要两台处理机顺序地不断地访问, 两者不会再发生冲突。这种寻址技术也可用于流水线多处理机中, 使各台处理机能彼此步调一致地存取向量数据。

如果争用是由于对一个或几个共享单元的访问而引起的, 那么需要采取另一种方法。

程序 7.1 是各处理机计算各自本地数据的和, 然后再将本地和相加以形成全程和的程序。假设各本地数据都存放在物理位置与各自处理机很近的局部存储器中, 所以计算本地和时存取本地数据不会出现争用。全程变量 Global-Sum 是所有数据的和。

我们通过并行地计算各个本地和, 然后把各个本地和加起来存入全程变量 Global-Sum 中, 这样来获得加速比。这和竞选过程非常相似, 首先各个选区计算出该选区的选票结果, 然后再报告给竞选中心统计出最后结果。现在的问题是最后共享数据的计算可能要用 $O(N)$ 时间, 因此它成为严重的系统瓶颈, 使得前面并行计算所获得的好处被抵消掉了。

程序 7.1

```

Procedure Add-to-Sum(VarGlobal-Sum;Real,Share d;Local-Table array of Real)
Var
i: integer;
Local-Sum: real;
Begin
  Local-Sum := 0.0
  For i := 1 to Max do

```

```

Local-Sum := Local-Sum + Local-Table[i];
LOCK(Global-Sum);
Global-Sum := Global-Sum + Local-Sum;
UNLOCK(Global-Sum);
end{ ProcedureAdd-to-Sum }

```

程序 7.1 中,本地计算操作把计算结果存入变量 Local-Sum 中,再将 Local-Sum 加到 Global-Sum 中。这里向共享全程变量加数时要特别小心。因此,我们应该提供一种机制使得当某台处理机读或写共享变量时,不应受其他操作的干扰。

例如,处理机 P_1 要向 Global-Sum 变量中加入数据 10,它首先读取 Global-Sum 的当前值,然后把 10 和这个当前值相加,最后把结果写回 Global-Sum。假如几台处理机同时进行加操作,那么全程变量的最后得到的和可能不正确。例如,全程变量 Global-Sum 的初值为 0,处理机 P_1, P_2 分别要向全程变量加入 10 和 15,操作过程如下:

1. 处理机 P_1 读取全程变量 Global-Sum 的值为 0。
2. 处理机 P_2 读取全程变量 Global-Sum 的值为 0。
3. 处理机 P_1 计算 Global-Sum 值,并把结果 15 存回全程变量 Global-Sum。
4. 处理机 P_2 计算 Global-Sum 值,并把结果 10 存回全程变量 Global-Sum。
5. 全程变量 Global-Sum 的最后值为 10。

显然这一过程出现了错误,处理机 P_1 向全程变量 Global-Sum 加入的 15 丢失了。处理机 P_2 从全程变量 Global-Sum 读到的值为 0,这是不正确的,应该是 15。当处理机 P_1 “拥有”全程变量 Global-Sum 期间,任何访问该变量的操作都应被禁止。也就是说,处理机 P_1 对于全程变量的读、改、写回的操作序列应该作为一个单独的原语操作,其执行期间不允许其他处理机访问该全程变量。程序 7.1 中语句 LOCK(Global-Sum) 和 UNLOCK(Global-Sum) 表示对全程变量(Global-Sum)的读、改、写操作是不允许中断的。

LOCK 语句的功能是如果该变量在当前没有被加锁,则该处理机继续执行后续的操作,否则该处理机就处于等待状态,直到此加锁变量被解锁为止。LOCK 语句的实现在硬件和软件两方面都应该十分仔细,因为它容易造成错误。错误地使用 LOCK 语句很容易出现死锁现象,这时两个或多个进程互相阻塞,除非有个进程解锁有关变量,否则所有进程都无法继续执行。而且由于进程都被阻塞,都无法执行程序中的 UNLOCK 语句。编程时如果错误地把某个变量上了锁,而无其他处理机能将它解锁,就将出现死锁。

编程时如果使用了 LOCK/UNLOCK 语句,无论 LOCK/UNLOCK 具体怎样实现,它都有可能成为系统的瓶颈。在总线互连的系统中,瓶颈很可能是总线。在交叉开关互连的系统中,通信的瓶颈消失了,系统性能将受到另一个系统瓶颈——共享存储器的影响。

程序 7.1 中 LOCK/UNLOCK 语句证明了共享存储器可能是系统的瓶颈。当然,用交叉开关代替总线的目的在于避免总线这个系统瓶颈,总线互连结构中 N 个并发总线请求需用时间是 $O(N)$ 。交叉开关互连把这个时间减少为 $O(1)$,但共享变量这个瓶颈仍需 $O(N)$ 时间。因此,使用交叉开关网络使一个程序的某些部分性能提高很多,而涉及共享变量的那些部分使系统操作效率很低。

以上我们主要讨论有关交叉开关互连的性能问题。另一方面,成本问题也很重要。交叉开关互连的成本与交叉点的数目成正比例,也即与 N^2 成正比例关系增加。总线互连的

成本与 N 成线性关系增加,它取决于总线接口的数目。当 N 很大时,交叉开关互连的成本非常高,以至于在整个系统的总成本中起决定作用。规模很大的交叉开关互连网络只有在每个交叉开关的成本非常低的情况下才是可行的。交叉开关互连网络的另一个问题是其有效的带宽往往不能得到充分利用,因此额外的开销只得到很低的效益。

C. mmp 计算机是采用交叉开关互连的例子。它由 Carnegie-Mellon 大学研制,从 70 年代初到 80 年代初使用了大约 10 年的时间。这种系统结构将 16 台 PDP-11/40 与 16 个存储器连接起来。从未打算将它做成商品化的样机,而只作为研究并行处理机应用及并行操作系统的工具。正因为这个原因,促使人们投入大量的人力与物力进行研究,获得了许多研究成果,为多处理机系统的进一步研究提供了许多经验。

采用交叉开关互连的主要目标是提高性能,但这并不是 C. mmp 的主要目标。如果把 16 台 PDP-11 处理机连接在一起完成某一任务以获得 16 倍的加速比,然而其总的速度比一台超高速的单处理机却还低得多,但 16 台 PDP-11 机器的价格比买一台 16 倍速度的单处理机要低。

C. mmp 系统的一个好处是它可以访问相当于一台 PDP-11 计算机存储空间 16 倍的存储器。因为存储器相对还比较贵,C. mmp 系统提供了在几个相对独立的进程之间分配昂贵资源的办法。C. mmp 系统的每台处理机有固定数量的本地存储器。大容量的共享存储器作为公共资源可以动态地分配给各个进程。C. mmp 系统还提供处理机池,可灵活、动态地分配给每个程序。理论上,16 台处理机可以分配给同一个程序,也可以把 5 台处理机分配给某一程序,把 3 台处理机分配给另一程序,直至处理机都被占用为止。

实际上,一个程序的每个进程常常需要很大的存储区,所以处理机台数小于 16 的多处理机系统的存储空间很快就会被分配完。好在 C. mmp 是一个用并行程序解决各类应用问题的实验系统。

我们很容易用带宽足够的其他互连结构来代替 C. mmp 系统中的交叉开关互连结构,系统的性能不会有本质上的差别。重要的一点是替代的互连结构的速度应足够快,能满足 C. mmp 系统的要求,决不能引入新的瓶颈,但新的互连结构的带宽不必和交叉开关的带宽相同。

C. mmp 系统说明了多处理机系统结构设计中的一个重要原则:系统的性能和价格是非常重要的因素,互连网络仅仅是系统的一个组成部分。如果系统结构设计者努力去掉通信中的瓶颈,那么这一努力可能将瓶颈移到系统的其他部分,而付出的代价却可能不合算。

就应用而论,重要的是首先假设系统的通信子系统的带宽是无限的且不存在争用现象,在这种情况下考虑应用问题能否在该多处理机系统上有效地运行。下一步是研究如何提供成本适当的通信网络,使其有限的带宽不使系统性能下降到某一规定值以下。

如果系统结构设计者在应用中发现某种方法存在固有的不可解决的问题,那么就应另找其他方法了。下一节将介绍另一种互连网络,它修改共享变量时可以避免串行操作。从这个意义上说,它是消除共享存储器这一系统瓶颈的唯一方法。

7.3.4 混洗交换互连和合并开关

混洗交换互连网络不但能用来连接向量处理机,例如用于循环归约和多重递归,还能用来连接相互独立的处理机。这一节将要讨论的混洗交换互连是与总线及交叉开关不同

的另一种互连方法,当然其带宽和成本也介于它们两者之间。

混洗交换网络能提供一种重要的功能,称为合并开关,它使某些操作在网络一级并行地执行,从而减少了争用现象,否则这些需要访问存储器的操作只能串行地执行。对那些需要互斥地访问共享变量的进程来说,这种方法很有潜力。

互斥地访问共享变量限制了大部分多处理机系统的性能,所以当出现多个进程要存取某个共享变量时,无论系统再增加多少台处理机也不可能再提高其速度。但是,对 RP3 系统和 Ultracomputer 系统来说,就没有这个问题。关于这两个系统下面会详细介绍。在这两个系统中,互斥访问工作一部分在通信网络中完成,另一部分在存储器中完成。实际情况是利用混洗交换网络提供的功能并行地而不是串行地完成互斥访问工作。

混洗交换网络能有效地支持互斥访问是有条件的,这些条件非常严格,有些应用问题可能不满足这些条件。这些应用问题都有由于访问共享变量而引起争用现象,从而成为系统的瓶颈。除非采用新的互连技术或者 N 很小,否则现有的多处理机的系统结构都不适于解决这些问题。

在图 7.42 所示的混洗交换网络中,一边是处理机,另一边是存储器。虽然从延迟角度看,存储器与处理机的距离比较远,但是各台处理机有很大容量的 Cache 存储器和局部存储器,所以大大降低了访问远程存储器的频率。

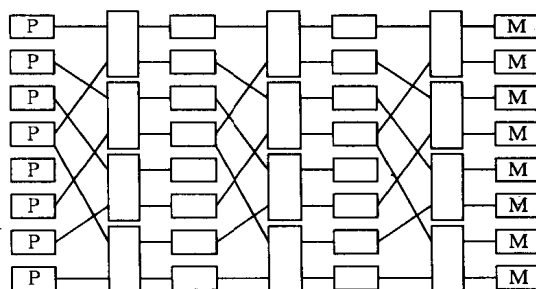


图 7.42 八台处理器和八个存储器通过混洗交换互连网络连接

重要的是图 7.42 所示的这种系统结构可以处理与总线结构及交叉开关结构相同的多处理机应用问题。混洗交换网络的带宽高于总线,而低于交叉开关。混洗交换网络的成本为 $O(N \log N)$,而总线和交叉开关的成本分别为 $O(N)$ 和 $O(N^2)$ 。

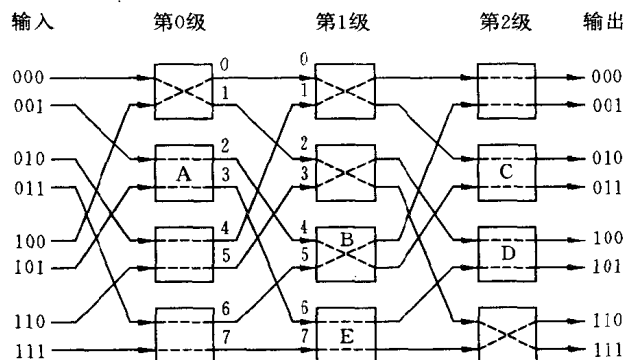
对于那些没有冲突的操作来说,混洗交换网络的带宽很高。如果 N 台处理机同时发出请求,例如 P_1 访问 M_{1+c} 单元的数据,那么这些请求可以同时被响应,而不会发生任何冲突。更一般的情况是如果 P_i 要访问 M_{i+c} 单元的数据,其中 P 是奇数, N 是 2 的幂,那么不会出现冲突。

我们假定处理机是互相独立的,并且无需同步,但许多实际应用问题要求处理机在某点同步后再继续执行。例如,绝大部分多处理机系统在执行 FFT 时总共有 $\log N$ 次重复,且下一次重复要在所有处理机完成本次重复后才开始,所以每次重复结束时有一个同步问题。

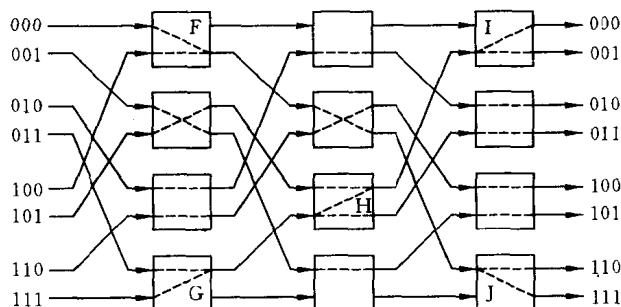
一旦各处理机被同步后,那么它们访问存储器的请求会同时发出。如果向量计算机访问一个向量的各个元素时很少出现争用,那么多处理机系统在同步后访问存储器的请求也不会出现过多的争用现象。

7.3.5 Omega 网络

这类网络已经用于伊利诺依大学的 Cedar 多处理机、IBM 的 RP3 和纽约大学 Ultra-computer。8 个输入端的 Omega 网络如图 7.43 所示。



(a) Omega 网络无阻塞地实现置换 $\pi_1 = (0, 7, 6, 4, 2) (1, 3) (5)$



(b) 置换 $\pi_2 = (0, 6, 4, 7, 3) (1, 5) (2)$ 在开关 F, G 和 H 上阻塞

图 7.43 由 2×2 开关组成 8×8 Omega 网络的两种开关设置

通常, n 个输入端 Omega 网络有 $\log_2 n$ 级, 从输入级到输出级依次编号为 0 到 $\log_2 n - 1$ 。通过检查二进制目的地址编码来控制数据路径。目的地址编码从高位开始的第 i 位为 0 时, 第 i 级的 2×2 开关的输入端与上输出端连接, 否则输入端与下输出端连接。

图 7.43(a) 和图 7.43(b) 的开关设置分别对应于置换 $\pi_1 = (0, 7, 6, 4, 2) (1, 3) (5)$ 和置换 $\pi_2 = (0, 6, 4, 7, 3) (1, 5) (2)$ 。

图 7.43(a) 中的开关设置可以实现 π_1 , 它的映射为 $0 \rightarrow 7, 7 \rightarrow 6, 6 \rightarrow 4, 4 \rightarrow 2, 2 \rightarrow 0, 1 \rightarrow 3, 3 \rightarrow 1, 5 \rightarrow 5$ 。观察一个消息从输入端 001 到输出端 011 的路径。它使用了开关 A、B、C。由于目的地址编码 011 的最高位“0”, 所以开关 A 设置成直送状态, 使输入端 001 与开关 A 的上输出端(编号为 2)连接。011 的中间一位是“1”, 开关 B 设置成交换状态, 使输入端 4 和下输出端连接。011 的最低位是“1”, 开关 C 设置成直送状态。同理, 开关 A、E 和 D 设置成对应状态, 形成一条消息从输入端 101 到输出端 101 的路径。实现图 7.43(a) 的置换 π_1 所要求的开关设置不存在冲突。

现在再来考察 8 个输入端口的 Omega 实现置换 π_2 (图 7.43(b)) 的情况。开关 F、G 和 H 的设置发生了冲突。F 产生冲突是由于 $000 \rightarrow 110$ 和 $100 \rightarrow 111$ 引起的, 因为两个目的地址编码的最高位都是“1”, 开关 F 的两个输入端都要求与下输出端相连。为了解决这一冲突, 必须拒绝一个请求。

同样地, $011 \rightarrow 000$ 和 $111 \rightarrow 011$ 引起开关 G 冲突, $101 \rightarrow 001$ 和 $011 \rightarrow 000$ 引起开关 H 冲突。开关 I 和 J 的设置能实现一个输入端与输出端相连的广播功能, 其条件是硬件开关要有四种合法状态。以上例子表明并不是所有的置换在 Omega 网络中一次通过便可以实现。

Omega 网络是一种阻塞网络。当出现阻塞时, 可以采用几次通过的方法来解决冲突。例如 π_2 , 在第一次通过时实现连接 $000 \rightarrow 110, 001 \rightarrow 101, 010 \rightarrow 010, 101 \rightarrow 001, 110 \rightarrow 100$ 。在第二次通过时实现连接 $011 \rightarrow 000, 100 \rightarrow 111, 111 \rightarrow 011$ 。通常, 如果采用 2×2 开关元件, n 个输入端 Omega 网络一次通过可以实现 $n^{n/2}$ 个置换, 但总共有 $n!$ 个置换。

$n=8$ 时, 意味着 8 个输入端的 Omega 网络一次通过只能实现全部置换的 10.16%, 即 $84/8! = 4096/40320 = 0.1016 = 10.16\%$ 。所有其他置换将引起阻塞。实现这些置换需要三次通过。一般来说, n 个输入端的 Omega 网络实现非阻塞连接最多需要通过的次数为 $\log_2 n$ 。在任何多级网络中都不希望出现阻塞, 阻塞将降低有效频宽。

如图 7.44(a) 所示, 采用上播或下播开关设置, Omega 网络也可以从一个源将数据广播给多个目的地。图 7.44(a) 中, 在输入端 001 的消息通过二进制树的连接广播到所有 8 个输出端。

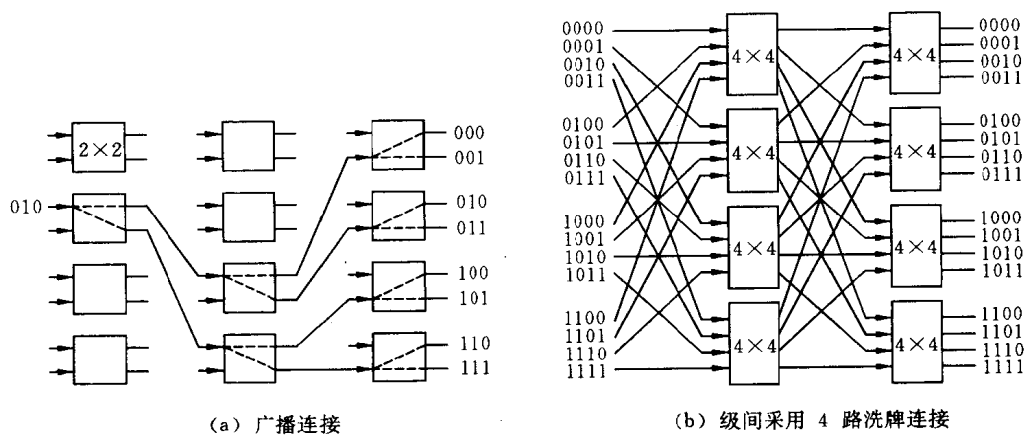


图 7.44 由 4×4 开关构成 Omega 网络的广播功能

图 7.44(b) 是用 4×4 开关元件作为构成块的 Omega 网络, 级间是 4 路洗牌连接, 而不是 2 路洗牌连接。16 个输入端的 Omega 网络, 级间是 4 路洗牌连接, 而不是 2 路洗牌连接。16 个输入端的 Omega 网络的级数为 $\log_4 16 = 2$ 。

4 路洗牌相当于 16 个输入端平均地分成 4 个子组, 然后 4 个子组均匀地洗牌。当采用 $k \times k$ 开关元件时, 则可以定义 k 路洗牌函数来构造更大的级数为 $\log_k n$ 的 Omega 网络。

图 7.45 是两个规模不同的蝶式网络。图 7.45(a) 是一个有 64 个输入端的蝶式网络,

它由 2 级($2=\log_2 64$) 8×8 交叉开关组成。0 级和 1 级之间采用 8 路洗牌连接。图 7.45(b)是有 512 个输入端的三级蝶式网络结构,同样也由 8×8 交叉开关构成。图 7.45(b)中每个 64×64 的方框相当于图 7.45(a)中的两级蝶式网络。

图 7.45(a)总共用了 16 个 8×8 交叉开关,图 7.45(b)共用了 $16\times 8+8\times 8=192$ 个 8×8 交叉开关。用这种模块结构构造更大的蝶式网络只要增加级数即可。蝶式网络不允许广播连接,所以蝶式网络是 Omega 网络的一个有限的子集。

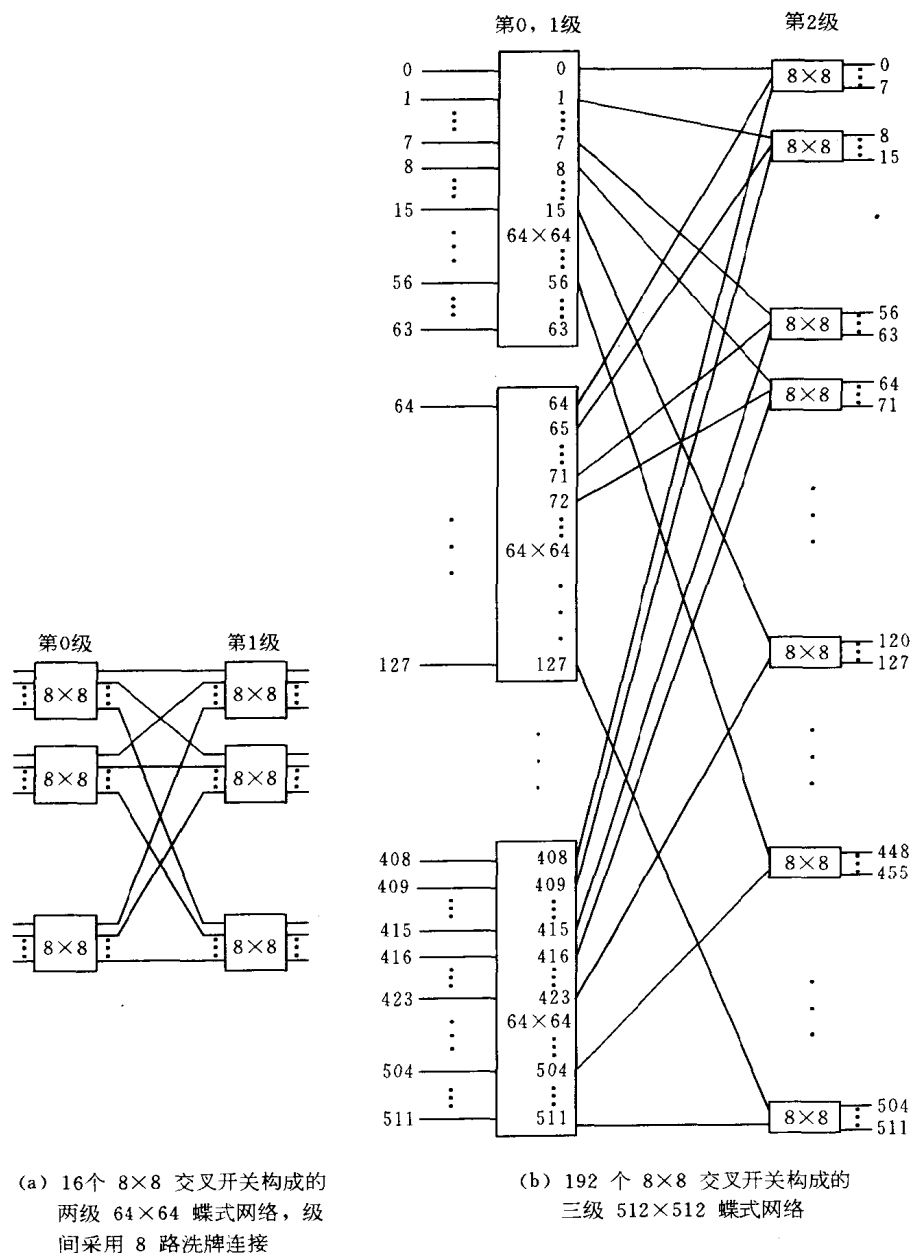


图 7.45 8×8 交叉开关构成模块结构的蝶式网络

7.3.6 蝶形操作

快速傅里叶变换 FFT 是数字信号处理中非常有效和有用的方法。计算 N 个复数点序列的有限离散傅里叶变换的常规算法,需要近似于 N^2 次复数乘法和 $N(N-1)$ 次复数加法。所以这种常规算法对信号作实时处理一般是有困难的。而 FFT 方法只需 $N\log_2 N$ 次运算。FFT 的基本思想是把 N 分解成大于 1 的整数的组合因子,例如把 N 分解为 P 个因子的乘积, $N = r_1 r_2 \cdots r_p$, 然后分别对长度比 N 小的 r_1, r_2, \dots, r_p 点序列进行有限离散傅里叶变换,从而使运算次数降为 $N(r_1 + r_2 + \cdots + r_p)$ 次。特别是当 N 是 2 的整数幂,即 $r_1 = r_2 = \cdots r_p = 2$ 时,运算次数可以进一步降为 $N\log_2 N$ 次,所以 FFT 的运算速度比常规算法约提高 $N/\log_2 N$ 倍。

多处理机上进行 FFT 运算需要两类处理机与处理机的通信。一类是蝶形操作,多对处理机交换数据并对其进行加权和差运算。另一类是二进制反向传输,它改变输出数据的次序。

上述两类操作与混洗交换操作是互不相容的。如果数据存储在各处理机,这种条件下蝶形操作不会有冲突,但二进制反向传输操作在互连网络上将出现非常严重的冲突。反过来,如果二进制反向传输操作没有冲突,那么蝶形操作将有很严重的冲突。

在网络中,上述两类操作至少有一类会出现冲突问题。一种典型的 FFT 实现需要对 N 个向量进行 $\log_2 N$ 次蝶形操作,紧接着或在 $\log_2 N$ 次蝶形操作之前进行一次二进制反向传输操作。因此,最好把数据分布在各个存储器中。这样,蝶形操作时不会出现冲突现象,而二进制反向传输会有冲突。

那么冲突有多严重呢? 最坏的情况是所有要访问的 N 项都在混洗交换网络的同一个结点的存储器中,那么得到这些数据所需时间为 $O(N)$, 这里我们假设数据很理想地分布在网络时,得到这些数据只需 $O(1)$ 时间。实际上 FFT 运算时二进制反向置换引起的冲突没有这么严重,因为蝶形操作一次就能访问 N 个不同的数据项,那么这些项一定分布在所有存储器中。

二进制反向传输操作是存取蝶形操作所存取过的这 N 项数据,所以冲突不会出现在存储器,而是在通信网络中。二进制反向置换时,最严重的冲突实际上只占 $O(N^{1/2})$ 时间,而不是 $O(N)$ 时间,所以可用带宽 $O(N)$ 中的 $O(N^{1/2})$ 被浪费了。

为了使数据通过混洗交换网络时不发生冲突,每个开关结点的两个输入数据必须分别输出到两个不同的输出端。如果两个输入数据都要输出到同一个输出端,则发生冲突。

置换访问操作时,网络的瓶颈是在该网络的中间这一级或中间两级。为了证明这个结论,考虑一种能引起最严重冲突的置换。在第一级,最坏情况是 $N/2$ 个开关结点中的每个开关的两个输入数据都要输出到同一个输出端。这就使得网络的第二级有一半开关的输入端是空的,另一半开关每个有两个输入数据。这种冲突情况在网络的第 2 级,第 3 级……,直至网络的中间一级都会出现。在中间这一级, $2^{(\log_2 N)/2}$ 个输入端的每个都有 $2^{(\log_2 N)/2}$ 个数据等待通过,而其余输入端都是空的。由于这些数据最终要送往位于网络末端的不同存储器,所以从瓶颈这一级开关开始,把数据向网络末端的通路进行分支。因此,在以后的每级开关,等待通过的数据个数每级开关减少一半,有数据的输入端增加一倍。最后,所有输入

端都有一个数据。

图 7.46 表示在有 16 台处理机和 16 个存储器的网络上的二进制反向传输。处理机 i 与存储器 i' 相连接, 这里 i' 是把二进制数 i 倒序得到的。因为 $(0010)=2$ 的倒序结果为 $(0100)=4$, 所以处理机 2 的目标是与存储器 4 相连。

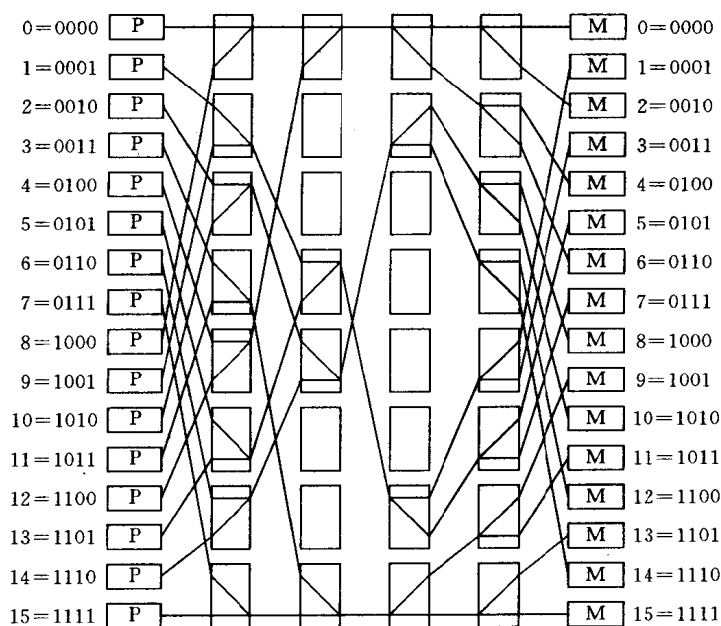


图 7.46 进行二进制反向传输的混洗交换互连网络
(网络中只有部分通路被使用)

关于混洗网络冲突的讨论告诉我们, 即使存储器没有冲突, 但由于存在通信冲突, 会有 $O(2^{(\log_2 N)/2}) = O(N^{1/2})$ 的延迟。采用交叉开关网络, FFT 不会遇到通信冲突和存储器冲突, 所以速度加快 $O(N^{1/2})$ 倍。

我们知道二进制反向传输的性能很差, 所以应设法去掉这种操作。例如, 在某些应用中, 我们采取如下处理步骤:

1. 使用 FFT 把时域变成频域。
2. 在频域中进行处理。
3. 使用 FFT 把频域变回时域。

这样, 在第一步结束时不必进行二进制反向传输操作, 而把二进制反向序直接输入到最后一步, 最后一步的 FFT 将产生一个次序的输出, 因此也不需要二进制反向传输, 这样把瓶颈避免了。

通常, 有必要找出通信网络中的冲突问题, 如果有可能, 应该采取办法把它解决。上面 FFT 就是这样一个例子, 它的瓶颈在一定条件下删除掉。当然我们不能保证瓶颈总能去掉, 但如果要删除它, 首先必须找到它。

如果处理机要处理分布在所有存储器上的数据, 那么通过访问共享控制变量与其他

处理机同步。我们对这种环境下开关的有效带宽很感兴趣。即使很简单的情况,例如假设访问地址均匀分布且不相关,计算有效带宽也比较困难。原因是缺少一个好的处理网络内部冲突的数学模型。当两个数据在某处冲突时,例如都请求某开关结点的同一个输出端输出,那么会发生什么情况呢?网络可以:

1. 随意地放弃一个,让另一个通过。
2. 一个在本地存储器中排队等待,另一个通过。
3. 当一个数据正在通过时,再来请求不予响应,这里假设被拒绝的请求发送者能够重发。

上述三种方法有一定的代表性,还有许多其他方法。有人用一个差分方程来描述删掉冲突信息后还剩下的信息个数,得到了非常令人满意的结果。他们发现有效带宽是 $O(N/\log_2 N)$, 而网络内部的冲突使带宽降到 $O(\log_2 N)$ 。通过排队论分析和模拟也得到了类似的结果。

一般分析没有把假定的输入与真实程序访问方式联系起来。周期地同步既有利又有害。同步将导致同一时刻大量访问网络。如果这些访问没有冲突,则是有利的,因为大量访问可在短时间内完成。但若这些访问的冲突很严重,则是有害的,因为它导致比静态方法预测要高的冲突率。

系统结构设计者不能想当然地认为平均带宽是 $O(N)$, $O(N/\log_2 N)$, $O(N^{1/2})$, $O(1)$, 而必须在实际应用中去开发网络的性能,建立符合实际应用中访问方式的可靠数学模型。对于这个问题, Pfister 和 Norton 在 1985 年写了一篇关于混洗网络热点冲突的文章。他们寻找当访问不完全均匀分布在整个存储器时混洗网络的有效带宽。他们的模型允许少部分访问特殊存储器,而大部分访问是均匀分布。他们的结果表明,有效带宽随访问相关性的增长而下降。在 Pfister-Norton 模型中,假设访问“热”存储器模块的概率是 h , 其他访问为均匀分布,当 N 台处理机的每台在每个周期访问存储系统 r 次时,“热”存储器模块接收请求的次数为:

$$\text{访问“热”存储器的请求次数} = r(1 - h) + rhN \quad (7.1)$$

其中第一项表示访问为均匀分布时,访问“热”存储器的次数。第二项表示访问“热”存储器的概率是 h 时访问的次数。因为存储器每个周期只能接收一个请求,所以式(7.1)左边的请求次数不能大于 1。因此,每台处理机访问存储器的最大有效请求率 R 等于式(7.1)达到 1 的比率,表示如下:

$$R = 1/(1 + h(N - 1)) \quad (7.2)$$

R 值随 N 的增大而减小。开关网络的有效带宽是式(7.2)给出的 R 值的 N 倍。

当 $h = 0$ 时,式(7.2)的值等于 1,则整个开关网络的带宽为 N 。当 h 值增大一点时,例如 1%,对 1024 台处理机来说,等于式(7.2)的分母值达到 11,有效带宽降到理想带宽的 $1/11$ 。甚至访问热存储器的概率很小,例如 0.1%,结果分母值为原来 2 倍,有效带宽降到原来的 $1/2$ 。

Pfister 和 Norton 通过模拟证明了他们的结论:冲突使网络出现如图 7.47 所示的树

形饱和。图 7.47 假定每个请求在得到满足之前能够保留,一个结点的内部队列长度可以是包括零的任何整数。

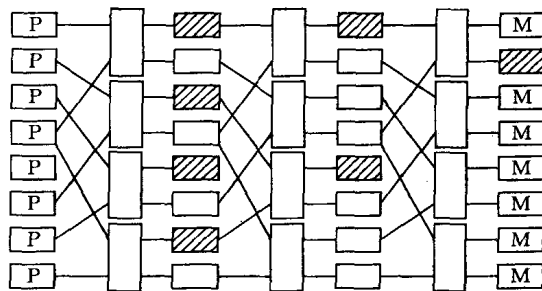


图 7.47 带阴影的是“热”存储器和处于饱和状态的开关

当“热”存储器不能接收新的数据时,送数据给它的开关结点就变成饱和状态,该开关结点的前面有关开关结点也将达到饱和状态。这样,前面有一系列开关结点将处于饱和状态。由于这些开关结点处于饱和状态,通信将受影响,系统的性能将迅速下降。在图 7.47 中,饱和结点用阴影表示,它们组成了一棵以“热”存储器为根的树。

如果一台处理机到一个存储器的路径中有些开关结点处于饱和状态,则发生阻塞。这种情况下的带宽要比等式(7.2)预计的低一些,它取决于饱和结点树的大小,也取决于每个结点中队列的大小。如果系统结构设计者想在网络中设置队列,图 7.47 表明为了减少热点竞争,最好将队列设在离存储器系统最近的那排开关处。当然队列也可设在其他地方,例如均匀地分布在整个开关网络,使得所有开关都一样了,以减少其他形式的竞争。

7.3.7 合并网络和取与加指令

在“热”存储器或网络的某个地方设置队列能消除最大负载所产生的影响,但队列不能缓和由于频繁访问存储器所引起的瓶颈问题,要解决这个问题,只有设法降低对热存储器的请求率。

Gottlieb 等人在 1983 年提出了一种很不平常的解决这一问题的方法,即在开关结点内部增加能实现计算的逻辑线路,以便减少访问共享变量存储单元的请求率。两个或多个访问同一共享单元的请求在某种条件下可以合并成一个,这样就减少了对共享存储单元的访问率,从而减少冲突和由于冲突而带来的带宽的下降。用这种系统结构的办法有时称为合并网络,它提供了一组新的指令,其中一条称为 Fetch-and-Add 指令,叫做取与加指令。

为了说明合并网络如何工作,让我们考察如图 7.47 通信网络中阴影结点所构成的树。树的根是一个存储器模块,它要接收比其他存储器模块更多的访问请求。在这个例子中我们列举一种可能出现的冲突情况,并说明 Fetch-and-Add 指令是如何解决这个问题的。

举一个排队问题的例子。假设 N 个请求每一个都要将一个数据项加到队列中。在传统方法中,队列指针不能被两个或更多个处理机同时修改,否则指针修改可能不正确。同理,对共享变量的并发累加也会出现错误结果。在程序 7.1 中我们采取顺序地修改的方

法,每个进程使用 LOCK 和 UNLOCK 操作互斥地访问全局变量。我们现在的解决办法是允许所有处理机或部分处理机同时修改队列指针。为做到这一点,我们使用如下定义的 Fetch-and-Add 指令:

```
Definition:Fetch-and-Add(Address,Increment);
Temp:=Memory[Address];
Memory[Address]:=Memory[Address]+Increment;
ReturnTemp;
```

当 M 台处理机并发地使用 Fetch-and-Add 指令时,需要满足下列条件:

1. 为了满足 M 个并发请求,Memory[Address]单元只能读写一次,而不能读写 M 次。
2. 并发地访问 Memory[Address]时返回给 M 个请求者的 M 个值的集合,应与按某顺序串行地互斥访问 Memory[Address]时返回给 M 个请求者的 M 个值的集合相同。

Fetch-and-Add 指令的操作很像 Add-to-Memory 指令的操作,唯一不同的是 Fetch-and-Add 指令返回值是操作之前的存储单元内容。

为了说明这个基本概念,我们考察 3 台处理机对一个存储单元 SUM 并行地执行 Fetch-and-Add 指令的情况。假设 SUM 的初值等于 10,三个增量分别是 2,5,12。网络产生一个增量总和 19,这样只需将它加到 SUM 上就可以了。访问 SUM 单元一次,以得到它的初始值 10。SUM 的新值是 $29=19+10$ 。同时,网络需要计算出返回给三个请求者的返回值,一种可能是 10,12,17,即依次用增量 2,5 和 12 计算所得的返回值。

具体的实现方法如图 7.48 所示。从图中可看到每个开关部件把要存入存储器的数据合并起来,把从存储器读出的数据做相反的操作。如果两条 Fetch-and-Add 指令的操作对象是同一个共享变量,那么当开关部件发现这两条指令同时出现在它的输入端时,把这两条指令的两个增量值相加产生一个和,把这个和存入存储器。这样,一个开关部件把 2 和 5 相加产生和 7,第二个开关部件把 7 和 12 相加产生和 19。

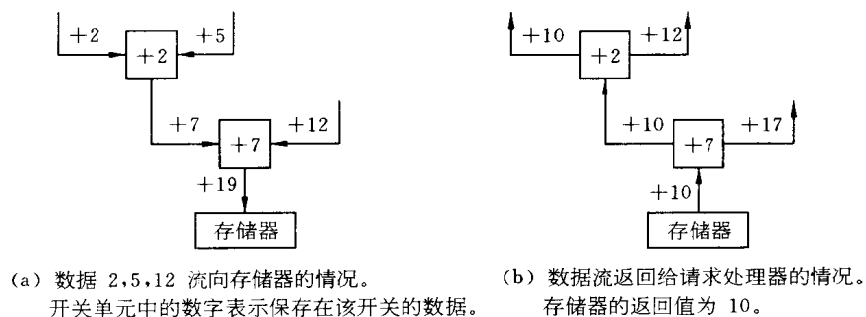


图 7.48 Fetch-and-Add 指令的实现方法

为了给出返回值,每个开关部件需要把两个增量中的一个存储起来,图 7.48 是存储左边输入的一个增量。因此,第一个开关部件存储 2,第二个开关部件存储 7。当 10 到达存储 7 的开关部件时,不作修改在它的左边输出端输出,同时将 10 和 7 相加的结果从右

边输出端输出,作为加 12 之前的值返回给请求者。10 送到第一个开关部件,不作修改在它的左边输出端输出,同时将 10 和 2 相加的结果从右边输出端输出。因此,左边输出端输出的 10 是 SUM 加 2 之前的值,右边输出端输出的 12 是 SUM 加 2 后但未加 5 之前的值。

每个合并开关部件至少要有以下功能:

1. 测试左、右输入的地址是否相同。
2. 两个增量相加。
3. 存储一个增量。

合并开关的这些功能的开销很大,但如果能删除某些共享变量的临界区从而减少热点竞争,那么合并开关还是合算的。

下面我们举一个使用 Fetch-and-Add 指令的具体例子。它是多处理机系统中的进队和出队问题。多处理机系统中一个常见的机制是当所有处理机都正在执行任务时,只能将新的任务暂存到队列中。当一台处理机完成当前的工作后,控制机制检查队列,如果队列中有任务,它就从队列中取出一个任务交给处理机去执行。

为了确保队列指针修改的正确性,必须有加锁和开锁操作,这样队列本身就成为瓶颈。例如,如果一个队列中有 N 个相互独立可立即执行的任务,当 N 台处理机同时完成任务可接收新的任务时,理想情况下,我们希望在一个周期内将 N 个任务分配出去,以便使所有处理机立即执行。然而,如果串行地修改队列指针,那么取出任务就要花费 $O(N)$ 时间。若 N 值很大,这个时间就很可观。如果任务本身很短,如只需 $O(1)$ 执行时间,那么取出任务的开销是无法让人忍受的。

这里所举的情况并非虚构。例如,写程序时常常加入阻塞语句。程序执行到阻塞语句时,处理机必须停止,直到所有处理机都通过阻塞点。一旦最后一个进程通过阻塞点,所有处理机同时变为自由。正常情况下,在阻塞点 N 台处理机同时被释放。这样,所有处理机都要从队列中得到新的任务,队列就成为严重的瓶颈。

使用 Fetch-and-Add 指令的基本思想是,想将一个数据项放入队列的每台处理机,在队列中申请到一个地址。这可用如下形式的语句实现:

```
enqueue-position := Fetch-and-Add(Head,1);
```

Fetch-and-Add 指令的第一个参数是一个计数器,它表示数据项放到队列的地址。第二个参数是一个增量值,当数据项放到队列后,Head 应加上这个增量值。

当程序串行执行时,Fetch-and-Add 指令将返回下一个要放入队列数据项在队列的地址。当两个或多个进程并行地执行时,所有 Fetch-and-Add 指令同时执行,每台处理机会获得数据项放入队列的正确地址值。并行执行 Fetch-and-Add 指令时的返回值与串行执行时的返回值完全相同。

这里没有讨论循环队列和队空或队满的情况,下一章要全面地讨论这些程序设计问题。上面所举的例子已充分说明使用 Fetch-and-Add 指令的优点。这是迄今为止为解决热点问题和消除多处理机程序中一系列瓶颈已被提出并正在认真地实现的唯一机制。

合并网络真能消除热点冲突吗?如果一个存储器接收到非常多的访问请求,那么这个

存储器就“热”了。只有当这些请求都访问同一个地址时,合并网络才能有效。这种情况现实吗?访问共享数据时会出现这种多个请求都是访问同一个地址的情况。

1985年IBM公司在RP3机器上探索了这个问题和其他许多有关的问题。RP3机器的结构如图7.49所示。左边是一台处理机,它是512台处理机中的一台,右边是由混洗交换网组成的合并网络。

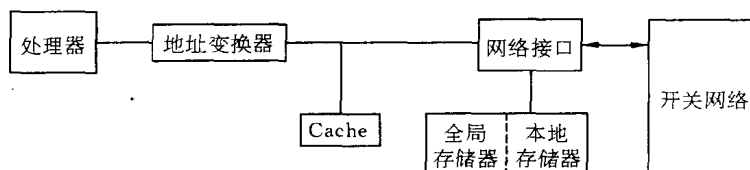


图 7.49 组成 IBM RP3 系统的处理器结构框图

(开关网络是带合并逻辑的混洗交换网络)

这个网络的输入与输出在同一侧。实际上,图7.47中的每个处理机结点和存储器结点完全相同。全局存储器分布在所有处理机中,这样每台处理机都有一个独立的存储块。每个存储块的一部分作为全局存储器,另一部分用作存储本地数据。在处理机与网络之间有一个地址变换器,一个Cache存储器和一个接口。接口负责把访问请求送往本地存储器或全局存储器或网络,通过网络把访问请求送到远程的全局存储器或远程的本地存储器。

这个系统的寻址方法很新颖。为了减少冲突,将全局地址空间均匀地分布在所有存储模块,使得对所有模块的请求比较平衡。这实现起来很容易,只需用存储器地址的低若干位来指定存储数据的模块,这样对逻辑地址空间中相邻数据的访问被均匀地分散到各个物理模块。

对本地存储器则不能这样处理。本地存储器的物理位置离它对应的处理机很近,用地址码的最高几位而不是最低几位来选择物理存储器。这样,本地存储器地址空间中相邻的数据应放在同一物理存储模块中。

RP3系统使用一个界处理这种困境,把有交叉地址的子空间和有块地址的子空间分开。如果有效地址处于这个界的上面,则地址码的最低几位决定物理模块,最高几位就是块内地址。如果有效地址处于这个界的下面,则地址码的最高几位决定物理模块,最低几位就是块内地址。前者的地址子空间用于共享的全局数据,后者的地址子空间用于本地数据。

本地数据不一定私有,因为有时一台处理机可能产生一个地址,这个地址属于远程处理机的本地地址空间。使用本地地址空间的主要目的是存于该空间的数据虽非共享但被频繁访问,所以要存于离处理机很近的地方。RP3系统的本地与全局子空间的分界线由软件控制,通过控制程序可以选择两个子空间的大小,而不是由硬件事先固定。

在结束这一节讨论时需要指出的是选择互连网络必须综合考虑速度和成本两个指标。总线的成本最低但速度最慢,交叉开关的成本最高但速度最快,混洗交换网络的速度和成本都介于总线和交叉开关之间。

混洗交换网络并不是速度和成本居于适中位置的唯一网络。如果增加扇入和扇出的花费不多,那么每个开关可以有较高的扇入和扇出系数,这样就可以减少通过网络的延

迟。基于这种基本原理的几种超立方体计算机系统已在 80 年代中期出现,其中并行性最高的是连接机(Connection Machine),它有 64 000 台处理机。最有影响是 Cosmic Cube 系统,它有 128 台处理机。虽然这些系统的设计没有采用合并开关,但超立方体连接方式是混洗交换网络的扩充。因此,也可以把混洗交换网络的合并开关的概念应用于超立方体网络。

总之,从总线到交叉开关以及中间各种网络,解决一个实际问题时,究竟用多少台处理机最有效?这取决于 R/C 比值的大小。总线方式的 R/C 值最低,当处理机数量增加时,总线的拓扑结构的效果可能最差。RP3 系统结构设法将本地数据和频繁使用的数据存放在一台处理机内,这样,增大了 R/C 比值和增加了使用效率较高的处理机台数。

目前,多处理机系统在商业上还处于初级阶段。以往的经验告诉我们:当前需要探索的主要未知领域是软件。解决各种实际问题最好的方法是将问题划分成诸模块,这些模块之间只需少量的通信,这样对互连拓扑的依赖性也就大大减小。相反,如果需要的通信次数不能减少,那么互连拓扑就变得非常重要。

习 题 七

7.1 解释下列术语

互连网络;互连函数;静态网络;动态网络;结点度;网络直径;等分宽度;共享介质网络;非阻塞网络;直接网络;间接网络;混合型网络;消息、包、片;存储转发寻径;虫蚀寻径;线路开关寻径;虚拟直通寻径;虚拟通道;缓冲区死锁;通道死锁;单播;选播;广播;会议;通道流量;网络通信时延

7.2 画出 16 台处理器仿 Illiac IV 的模式进行互连的互连结构图,列出 PE。分别只经一步、二步和三步传送就能将信息传送到各处理器号。

7.3 设 16 个处理器编号分别为 0、1、…、15,要用单级互连网络。当互连函数分别为

(1) $Cube_3$

(2) $PM2_{+3}$

(3) $PM2_{-0}$

(4) Shuffie

(5) Shuffie (Shuffle)

时,第 13 号处理器各与哪一个处理器相连?

7.4 在编号分别为 0、1、2、…、F 的 16 个处理器之间,要求按下列配对通信:

(B,1),(8,2),(7,D),(6,C),

(E,4),(A,0),(9,3),(5,F)。

试选择所用互连网络类型、控制方式,并画出该互连网络的拓扑结构和各级交换开关状态图。

7.5 画出编号分别为 0、1、…、F 共 16 个处理器之间实现多级立方体互连的互连网络,当采用级控制信号为 1 100(从右至左分别控制第 0 级至第 3 级)时,9 号处理器连向哪个处理器?

- 7.6 对于采用级控制的三级立方体网络,当第 i 级 ($0 \leq i \leq 2$) 为直连状态时,不能实现哪些结点之间的通信?为什么?反之当第 i 级为交换状态呢?
- 7.7 假定 8×8 矩阵 $A = (a_{ij})$, 顺序存放在存储器的 64 个单元中,用什么样的单级互连网络可实现对该矩阵的转置交换? 总共需要传送多少步?
- 7.8 假定有 128 个处理器,采用 PM2I 多级网络完成某种变换,若 $i = 2$ 的一级损坏,今拟用 $Cube_i$ 网络代替损坏的这一级,试说明最多要几级?
- 7.9 用单级立方体网络模仿 $N = 16$ 的单级 PM2I ($i = 0$) 网络,最差情况下要用几次单级循环传送?
- 7.10 试拟定用混洗交换网络来模拟立方体单级网络的算法,并求出模拟步数的上、下限值。
- 7.11 分别使用立方体单级网络和混洗交换单级网络将一个 PE 的数据播送到所有 2^n 个 PE 中去,问各需要循环传送多少步? 其中假设混洗交换单级网络每步只能进行混洗或交换中的一种变换,立方体单级网络第 i 步完成 $Cube_i$ 的变换传送。
- 7.12 并行处理机有 16 个处理器,要实现相当于先 4 组 4 元交换,然后是 2 组 8 元交换,再次是 1 组 16 元交换的交换函数功能,请写出此时各处理器之间所实现之互连函数的一般式,画出相应多级网络拓扑结构图,标出各级交换开关的状态。
- 7.13 给出 $N = 8$ 的蝶式交换如图 7.50 所示。
 (1) 写出互连函数关系式;
 (2) 如采用 Omega 网络需几次通过才能完成此交换;
 (3) 列出 Omega 网络实现此交换的控制状态图。
- 7.14 具有 $N = 2^n$ 个输入端的 Omega 网络,采用单元控制, 图 7.50 习题 7.13 附图
 (1) N 个输入总共应有多少种不同的排列?
 (2) 该 Omega 网络通过一次可以实现的置换总共可有多少种是不同的?
 (3) 若 $N = 8$, 计算出一次通过能实现的置换数占全部排列的百分比。
- 7.15 画出 $N = 8$ 的立方体全排列多级网络,标出采用单元控制实现 $0 \rightarrow 3, 1 \rightarrow 7, 2 \rightarrow 4, 3 \rightarrow 0, 4 \rightarrow 2, 5 \rightarrow 6, 6 \rightarrow 1, 7 \rightarrow 5$ 的同时传送时的各交换开关状态。说明为什么不会发生阻塞?
- 7.16 分别画出 4×9 的一级交叉开关以及用两级 2×3 的交叉开关组成的 4×9 的 Delta 网络,比较一下交叉开关设备量的多少。
- 7.17 画出用 4×4 交叉开关组成一个 3 级的 16×16 交叉开关网络,其设备量比单级 16×16 的交叉开关节省多少设备? 举例说明在输入和输出之间存在着较多的冗余路径。
- 7.18 说明 4×4 交叉开关组成的两级 16×16 交叉开关网络虽节省了设备,但它是一个阻塞式网络。
- 7.19 图 7.51 是一个 $2^3 \times 2^3$ 的 Delta 网络。
 (1) 问该网络在任何处理机和任何存储器模块之间是否都有一个通路?
 (2) 令 $d_2 d_1 d_0$ 是二进制编号为 $P_2 P_1 P_0$ 的某处理机所要访问的存储模块号的二进制

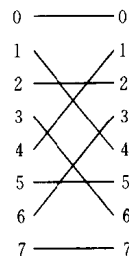


图 7.50 习题 7.13 附图

编码,网络中第 0、1、2 级的控制信号分别为 x_0 、 x_1 、 x_2 ,其中第 i 级控制信号 x_i 为 0 时,控制为直连, x_i 为 1 时控制为交叉连接。根据某处理机 $P_2P_1P_0$ 给出的访存模块号 $d_2d_1d_0$,为了将网络通路建立起来,请写出控制信号 x_0 、 x_1 、 x_2 与 d_0 、 d_1 、 d_2 及 P_0 、 P_1 、 P_2 的逻辑关系式。

- (3) 若 0 号处理机访问 2 号存储模块的同时,4 号处理机要访问 4 号存储模块,6 号处理机要访问 3 号存储模块,问是否发生阻塞?

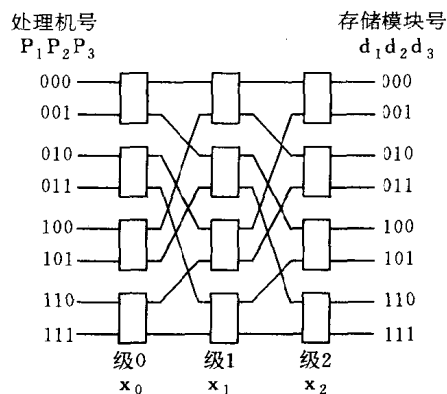


图 7.51 习题 7.19 附图

- 7.20 令 $2^m \times 2^m$ 矩阵 A 以行主方式存放在主存储器中,试证明在对 A 进行 m 次完全混洗变换后可获得转置矩阵 A^T 。
- 7.21 (1) Benes 二元网是一种重安排和非阻塞多级网络,因为重安排它已有的连接可以在输入和输出间实现全部可能的连接,因而总能给新的输入-输出对建立一条新的通路。

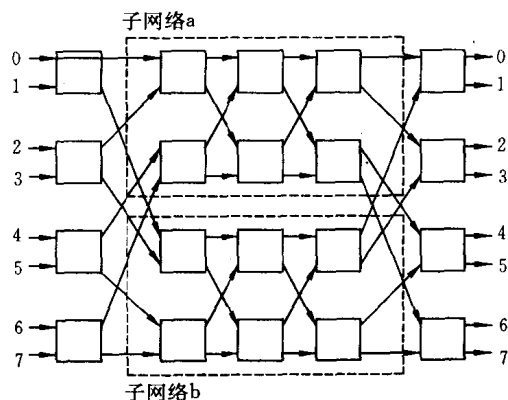


图 7.52 习题 7.21 的 8×8 Benes 网络

试设计一寻径算法在 8×8 的 Benes 网络上实现下列排列:

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 7 & 4 & 0 & 2 & 6 & 1 & 5 \end{pmatrix}$$

在算法进行了第一次迭代后的输入输出开关元件控制状态的设置如图 7.52 所示, 要求所设计的算法可以递归地施用于图中标以 a 和 b 的两个中间级子网。

(2) 将已经学过的 SIMD 多级互连网络按阻塞、重安排和非阻塞三种不同的特征进行分类。

7.22 设 N 个输入端的 Omega 网络 ($N = 2^n$), 它的每个开关单元都是独立控制的。

(1) 给定任意一个源-目的(S-D)对, 其连接通路可用目的地址唯一控制, 现不用目的地址(D)作为寻径标记, 而定义 $T = S \oplus D$ 作为寻径标记。试说明可以单独用 T 来确定连接通路。用 T 作为寻径标记的优点是什么?

(2) Omega 网络能实现(一源到多目的)播送功能, 若目的 PE 数为 2 的幂, 试给出一简单的寻径算法以完成这一功能。

7.23 在下列单级互连网络中, 将信息从一个 PE 播送给所有其他 PE 要用多少步 ($N = 2^n$ 个 PE)?

(1) 混洗交换网络, 每步只能做一次混洗或一次交换, 但不能两者混合。

(2) 立方体网络, 每步 i ($0 \leq i \leq n-1$) 可实现寻径函数 C_i 。

7.24 试证明一次通过 Omega 网络能不能实现任意的移数排列。所谓移数排列可定义如下: 给定 $N = 2^n$ 个输入端, 将数据循环左移或循环右移 k ($0 \leq k < N$) 个距离即为移数排列。

7.25 试证明多级 Omega 网络采用不同大小构造块构造时所具有的下列特性。

(1) 一个 $k \times k$ 开关模块的合法状态(连接)数目等于 K^k 。

(2) 试计算 2×2 开关模块构造 64 个输入端的 Omega 网络一次通过所能实现置换的百分比。

(3) 采用 8×8 开关模块构造 64 个输入端 Omega 网络, 重复(b)。

(4) 采用 8×8 开关模块构造 512 个输入端的 Omega 网络, 重复(b)。

7.26 (1) 画出 2×2 开关构成的 16 个输入端的 Omega 网络。

(2) 结点 1011 传送消息给结点 0101, 同时结点 0111 传送信息给结点 1001, 画出完成这一寻径的开关设置。这种情况会出现阻塞吗?

(3) 试计算这个 Omega 网络一次通过实现的置换个数, 一次通过实现的置换个数占全部置换的百分比为多少?

(4) 这个网络实现任意一个置换最多的通过次数是多少?

7.27 试确定下列网格计算机和超立方体多计算机中的最优寻径路径。

(1) 假设有一个 64 个结点的超立方体网络, 根据 E 立方体寻径算法, 画出从结点 101101 发送消息给结点 011010 的路径, 并标出这条路径上的所有中间结点。

(2) 在一个 8×8 网格上, 根据下面的条件确定两条优化的选播路径, 源结点是 (3,5), 10 个目的结点是 (1,1), (1,2), (1,6), (2,1), (4,1), (5,5), (5,7), (6,1), (7,1), (7,5)。

(i) 第一条选播路径应使通道数最少。

(ii) 第二条选播路径应使从源结点到每个目的结点的距离最短。

(3) 假设有一个 16 个结点的超立方体网络, 源结点是 (1010), 9 个目的结点是

(0000), (0001), (0011), (0100), (0101), (0111), (1111), (1101), (1001)。根据贪婪算法, 尽可能地使用流量较少的通道, 确定一条较优的选播路径, 使从源结点到所有目的结点的距离最短。

- 7.28 根据推理、分析或反例证明下列命题正确。
- (1) 采用虫蚀寻径的超立方体多计算机系统, 相邻结点之间有一对方向相反的单向通道, 试证明在该系统上实现 E 立方体寻径不会死锁。
 - (2) 试证明在 2 维网格上实现 X-Y 寻径不会死锁。
 - (3) 试证明在 3 维网格 (k 元 n 方体) 上实现 E 立方体寻径不会死锁, 该方法已用于采用虫蚀寻径和阻塞流控制方法的 J-Machine 系统。
- 7.29 给定一采用完全混洗互连网络, 并有 256 个 PE 的 SIMD 机。假如执行混洗互连函数 10 次, 则原来在 PE_{123} 中的数据将被送往何处。

第八章 SIMD 计算机

SIMD 计算机有时也称并行处理机。因为它是以单一控制部件控制下的多个处理单元构成的阵列,所以有时也称为阵列处理机。SIMD 计算机主要使用于要求大量高速向量或矩阵运算的场合。本章讨论 SIMD 计算机的模型、基本结构、并行算法和具体实例。

8.1 SIMD 计算机模型

在第一章我们给出过单指令流多数据流 SIMD 计算机的抽象模型。它在同一个控制部件管理下,有多个处理单元。所有处理单元均收到从控制部件广播来的同一条指令,但操作对象是不同的数据。下面我们介绍 SIMD 计算机的操作模型。

H. J. Siegel 提出了 SIMD 计算机的操作模型,如图 8.1 所示。

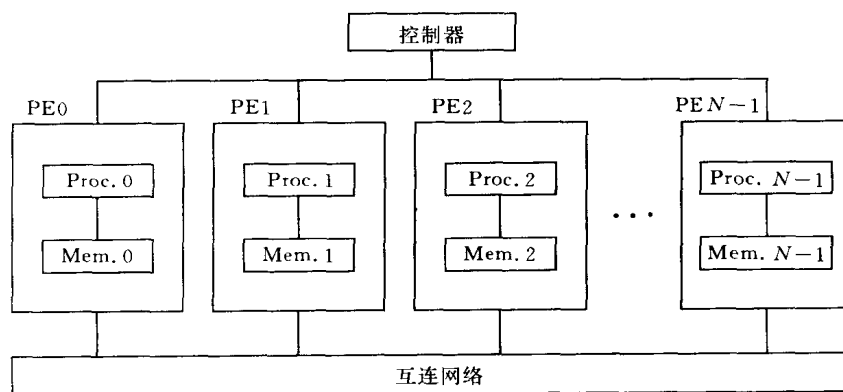


图 8.1 SIMD 计算机的操作模型

SIMD 计算机的操作模型可用五元组表示:

$$M = (N, C, I, M, R)$$

式中:

(1) N 为机器的处理单元(PE)数。例如, Illiac IV 有 64 个 PE。而连接机(Connection Machine)CM-2 采用 65536 个 PE。

(2) C 为由控制部件(CU)直接执行的指令集,包括标量和程序流控制指令。

(3) I 为由 CU 广播至所有 PE 进行并行执行的指令集,它包括算术运算、逻辑运算、数据寻径、屏蔽以及其他由每个活动的 PE 对它的的数据所执行的局部操作。

(4) M 为屏蔽方案集,其中每种屏蔽将 PE 集划分为允许操作和禁止操作两种子集。

(5) R 是数据寻径功能集,说明互连网络中 PE 间通信所需要的各种设置模式。

我们可以用上述五元组描述一台具体的 SIMD 机器,如 MasPar MP-1 计算机的操作特性如下:

- (1) MP-1 是一种 SIMD 机器,其 PE 数 $N = 1\,024$ 至 $16\,384$ 。PE 数目与机器配置有关。
- (2) CU 执行标量指令,将译码后的向量指令广播到 PE 阵列,并控制 PE 间的通信。
- (3) 每个 PE 都是基于寄存器的加载/存储 RISC 处理机,能执行不同数据量的整数运算和标准浮点运算。PE 从 CU 接受指令。
- (4) 屏蔽方案设在每个 PE 中,并由 CU 连续监控,它能在运行时动态地使每个 PE 处于置位或复位状态。
- (5) MP-1 有一个 X-Net 网络和一个全局多级交叉开关寻径器,以实现 CU-PE 之间、X-Net 的 8 个近邻之间和全局寻径器的通信。

表 8.1 列出了三种 SIMD 计算机。这些系统的 PE 数从 DAP 610 的 4096 到 MasPar MP-1 的 16 384 和 CM-2 的 65 536。CM-2 与 DAP 610 都是细粒度、位片式 SIMD 计算机。PE 都带有浮点加速器。

表 8.1 典型 SIMD 计算机

系统型号	SIMD 计算机系统结构和性能	语言、编译器和软件支持
MasPar 计算机公司 MP-1 系列	可用的配置为 1 024 至 16 384 台处理机,达 26 000 MIPS 或 1.3 Gflops。每个 PE 是 RISC 处理机,带 16K 字节本地存储器。X-Net 网络加一个多级交叉开关互连网。	Fortran77、MasPar Fortran (MPF) 和 MasPar 并行应用语言;X-窗口、UNIX/OS,符号调试程序,可视化和动画片制作器。
Thinking Machines 公司 CM-2	多至 65 536 PE 位片阵列排成 10 维超立方体,每个顶点为 4×4 网格,每个 PE 最多有 1M 位存储器,32 PE 的模块之间共享 FPU 选件,峰值速度 28 Gflops 和持续速度 5.6 Gflops。	由 VAX, Sun 或 Symbolics 360 主机驱动,PARIS 支持的 Lisp 编译器、Fortran90、C* 和 *Lisp。
Active Memory Technology DAP 600 系列	1K 位/PE 方形网格互连成 4 096 PE 的细粒、位片 SIMD 阵列,正交 4-邻位链接,20 GIPS 和 560 Mflops 峰值性能。	由主机 VAX/VMS 或 UNIX Fortran-plus 或 DAP 上 APAL 提供,主机的 Fortran77 或 C。与 Fortran90 标准有关的 Fortran-plus。

MP-1 的每个 PE 装有 1 位逻辑单元、4 位整数 ALU、64 位尾数部件和 16 位指数部件。所以,MP-1 是中粒度 SIMD 机。由于每个 PE 比较简单,故可将几个 PE 做在一个芯片上。MP-1 的每个芯片有 32 个 PE,每个 PE 有 40 个 32 位寄存器。32 个 PE 用 X-Net 网络互相连接,这是一种对角线双级链路扩展的 4-邻网。

CM-2 在一个芯片上将 16 个 PE 连成网络网络,每个 16-PE 网的芯片安置在 12 维超立方体的顶角上。这样, $16 \times 2^{12} = 2^{16} = 65\,536$ 个 PE 组成了整个 SIMD 阵列。

DAP 610 则在一个芯片上实现了一个 64-PE 的网络网络,再由这些芯片上小网格互相连接组成一个大网格 (64×64)。目前,Fortran 90、C 的各种修订文本、Lisp 和其他同步程序设计语言都已研制成功,可为 SIMD 机编程序用。

8.2.2 共享存储器结构

共享存储器结构的 SIMD 计算机如图 8.3 所示。这是一种集中设置存储器的方案。共享的多体并行存储器 SM 通过对准网络与各处理单元 PE 相连。存储模块的数目等于或略大于处理单元的数目。同时在存储模块之间合理分配数据,通过灵活、高速的对准网络,使存储器与处理单元之间的数据传送在大多数向量运算中都能以存储器的最高频率进行,而最少受存储冲突的影响。这种共享存储器模型在处理单元数目不太大的情况下是很理想的。Burroughs Scientific Processor (BSP)采用了这种结构。16 个 PE 通过一个 16×17 的对准网络访问 17 个共享存储器模块。存储器模块数与 PE 数互质可以实现无冲突并行地访问存储器。

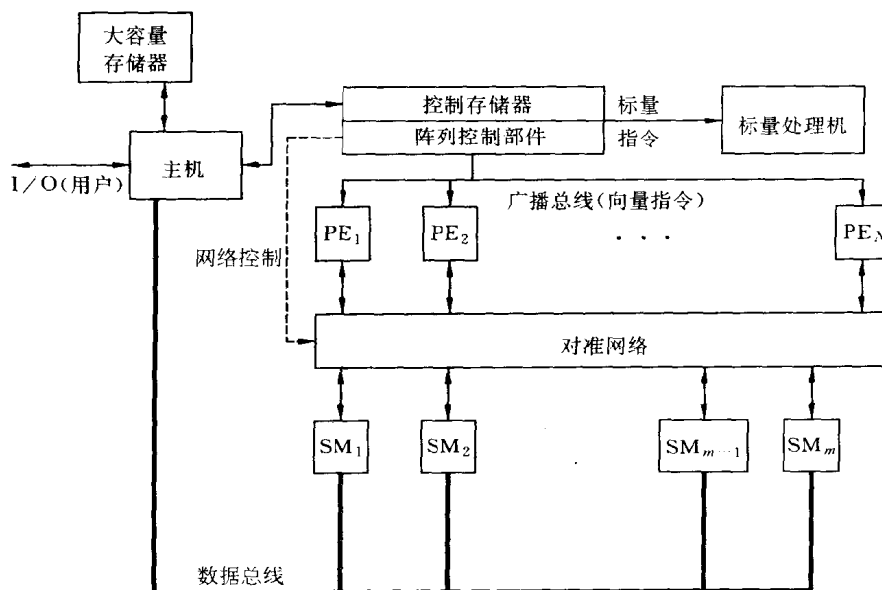


图 8.3 共享存储器的 SIMD 计算机

无论采用哪种存储方案,互连网络的存在都是必要的。在共享内存方案中,它是内存与处理单元之间的必由之路。在分布内存方案中,即使处理单元所需数据在大多数情况下能由本地存储器提供,处理单元之间的数据交往仍是必不可少的。在图 8.2 中,各处理单元 PE 之间可以经过两条途径相互联系:一条是直接通过数据寻径网络;另一条是数据从 LM 读至阵列控制部件,然后通过公共数据总线“广播”到全部 PE 中。在处理单元数目很多的并行处理机中,PE 之间的直接数据通路只能是很有限制的,这决定了系统的固定结构和专用处理机的性质。这种局限性需要从互连网络的研究中得到解决。

SIMD 计算机执行向量指令,对向量的分量进行算术、逻辑、数据寻径和屏蔽操作。在位片 SIMD 计算机中的向量是二进制向量。在字并行 SIMD 计算机中向量的分量是 4 字节或 8 字节的数。

所有 SIMD 指令都必须使用长度为 n 的向量操作数,其中 n 是 PE 的个数。SIMD 指

令与流水线向量处理机的指令类似,不同之处是多 PE 的空间并行性代替了流水线的时空并行性。

数据寻径指令包括置换、广播、选播以及多种循环和移数操作。在任何指令周期,通过屏蔽操作可以允许或禁止某些 PE 参加运算。

上述 SIMD 结构的所有 I/O 动作都是由主机处理的。在主机和阵列控制部件之间有一个专用的控制存储器,它是一个存放程序和数据的中间存储器。

在启动程序之前,把划分好的数据集合分布到本地存储器或共享存储器模块。主机管理大容量存储器或计算结果的图形显示。在控制部件的协调下,标量处理机与 PE 阵列并发地运算。

8.2.3 SIMD 计算机的特点

在第六章我们介绍了流水线向量处理机。向量处理机和 SIMD 计算机都能对大量数据进行向量处理,但它们之间又有很大的区别,SIMD 计算机具有以下一系列特点。

SIMD 计算机与流水线向量处理机一样,特别适于高速数值计算。它是以诸如有限差分、矩阵、信号处理、线性规划等一系列计算问题为背景而发展起来的。这些问题的共同特点是循各种途径归结为数组和向量处理。与按多指令流多数据流方式工作的多处理机相比,SIMD 计算机具有较固定的结构,它直接与一定的算法相联系,其效率取决于计算程序向量化的程度。它可以在这个前提下,通过改进系统结构和制定并行算法,使可能适应的计算问题类型尽量广一些、多一些。这就是说,应该把系统结构的研究和算法的研究结合起来。

SIMD 计算机利用大量处理单元对向量所包含的各个分量同时进行运算,这正是它获得很高处理速度的主要原因。与同样擅长于向量处理的流水线向量处理机相比,SIMD 计算机依靠的并行措施是资源重复,而不是时间重叠;而且它的每个处理单元要担负多种处理功能,相当于流水线向量处理机的多功能流水线部件(如在 TI ASC 机中),其效率比多个单功能流水线部件(如在 CRAY-1 机中)要低一些。所以,只有在硬件价格大幅度下降,加上系统结构的不断改进,SIMD 计算机才具有较好的性能价格比。但是,要论提高运算速度,SIMD 计算机主要依靠增多处理单元的个数,与流水线处理机主要依靠缩短时钟周期相比,其提高速度的潜力要大得多;如果有很好的互连网络相配合,则多处理单元的功能和灵活性将会更强一些。时钟周期为 160ns、包含 16 个处理单元的 BSP SIMD 计算机取得的运算速度,能够与时钟周期为 12.5ns、包含 12 条单功能流水线的 CRAY-1 流水线向量处理机相当,这就是一个很好的例证。

SIMD 计算机与流水线向量处理机的另一区别是它的互连网络,这是由多处理单元这一特点所决定的。虽然目前的 SIMD 计算机采用的互连网络还比较简单,但它毕竟是 SIMD 计算机最有特色的一个组成部分。正是它规定了处理单元的连接模式,决定了 SIMD 计算机能适应的算法类别,对整个系统的各项性能指标产生了重要的影响,因此它成为 SIMD 计算机结构的研究重点。同时,它也是多处理机的重要组成部分。

对于以向量处理为主的 SIMD 计算机而言,除向量运算速度以外,整个系统的实际有效速度还在很大程度上取决于另外两个因素:一是标量运算速度;二是编译过程的开销。

即使是向量运算,向量长度的影响也是不可忽视的。流水线的向量处理机处理短向量时,流水线建立和排空时间的比例加大;而在 SIMD 计算机中,短向量对速度的影响虽较小,但也降低了处理效率。试设想:如果某一台机器的向量处理速度极高,甚至是不受限制的,但标量处理速度只是每秒一百万次浮点运算,那么对于标量运算占 10% 的题目来说,总的有效速度就不过是每秒一千万次浮点运算。由此可见,提高 SIMD 计算机处理标量和短向量的能力是很重要的。至于编译时间,它既与系统结构,又与机器语言的水平高低有密切的关系。特别是如果要提高 SIMD 计算机的通用性,则建立一个具有向量化功能的高级语言编译程序是十分必要的。

SIMD 计算机还有一个重要特点,这就是,它基本上是一台向量处理专用计算机。尽管它有一个功能很强的控制部件实际上起着标量处理机的作用,但仍然必须和一台高性能单处理机配合工作,使后者担负系统的全部管理功能。从这个意义上来看,它们根据功能专用化的原则组成一个异构型多计算机系统,向量处理部件是系统的主体,而高性能单处理机可视为它的前端机,用来分担部分功能,以便充分发挥主体的向量处理效率。流水线向量处理机则有一些不同:除了 CDC STAR100 和 CRAY-1 等巨型计算机本身被认为是完整的通用系统以外,还有 AP120B、IBM 3838 等专用浮点数组处理机,它们接到主机上是为了执行主机的一些有关操作或子程序,以此分担主机的部分功能,从而提高主机的有效运算速度。在这样的系统中,AP102B、IBM 3838 等只起着后端处理机的作用,它们还不能被认为是系统的主体。

8.3 SIMD 计算机实例

SIMD 计算机的发展过程如图 8.4 所示。

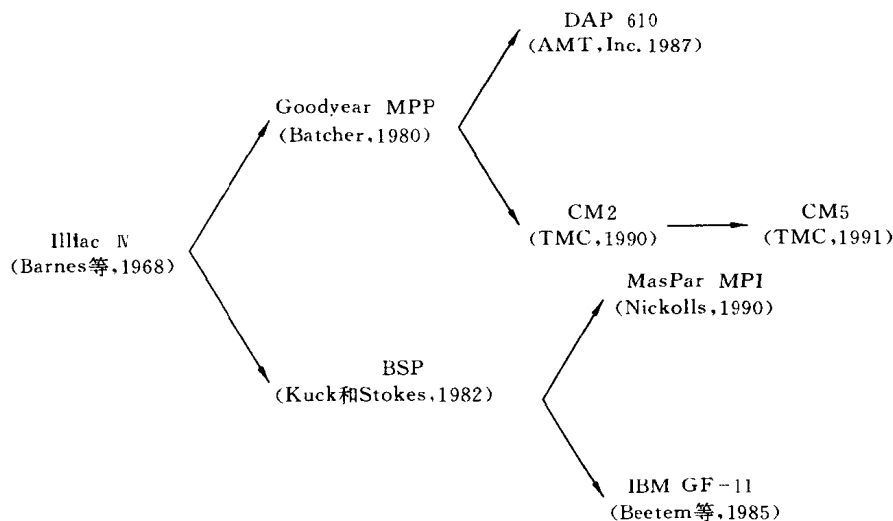


图 8.4 SIMD 计算机发展过程

Illiad IV 是最先采用 SIMD 计算机结构的计算机。随后一个方向是用位片 PE 制造的

SIMD 计算机,如 Goodyear MPP、AMT/DAP610 和 TMC/CM-2。CM-5 是以 SIMD 模式运行的同步 MIMD 计算机。另一个方向是用字宽运算 PE 的中粒度 SIMD 计算机。BSP 是 16 台处理机和 17 个存储模块同步工作的共享存储 SIMD 计算机。GF-11 是由 IBM Watson 实验室研制、作科学模拟研究用的。MasPar MP-1 是中粒度 SIMD 计算机。下面我们将介绍 Illiac IV、BSP 和 MasPar MP-1。

8.3.1 Illiac IV 阵列处理机

Illiac IV 阵列处理机由美国宝来公司和伊利诺依大学 1965 年开始研制,并于 1972 年生产的。

Illiac IV 系统的原理总框图如图 8.5 所示。总起来说,它由两大部分组成,即 Illiac IV 阵列和 Illiac IV 输入输出系统。实际上,它是由三种类型处理机联合组成的多机系统:一是专门对付数组运算的处理单元阵列(processing element array);二是阵列控制器(array control unit),它既是处理单元阵列的控制部分,又可以视为一台相对独立的小型标量处理机;三是一台标准的 Burroughs B6700 计算机,担负 Illiac IV 输入输出系统和操作系统管理功能。

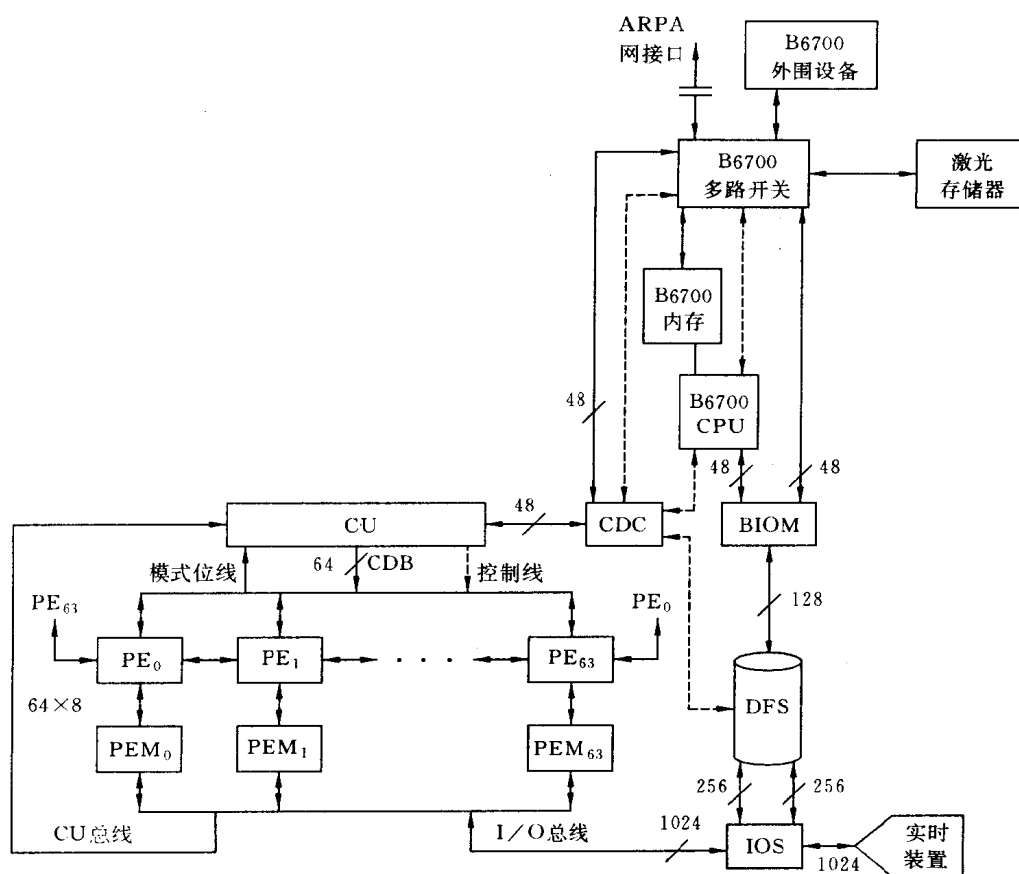


图 8.5 Illiac IV 系统总框图

1. Illiac IV 阵列

Illiac IV 阵列由 64 个处理单元、64 个处理单元存储器和存储器逻辑部件所组成。这个阵列的 64 个处理部件 $PU_0 \sim PU_{63}$ 排列成 8×8 的方阵, 每一个 PU_i 只和其东、西、南、北四个近邻 $PU_{i+1} \pmod{64}$ 、 $PU_{i-1} \pmod{64}$ 、 $PU_{i+8} \pmod{64}$ 和 $PU_{i-8} \pmod{64}$ 有直接连接。循此规则, 南北方向上同一列的 PU 两端相连成一个环, 东西方向上每一行的东端 PU 与下一行的西端 PU 相连, 最下面一行的东端 PU 则与最上面一行的西端 PU 相连, 从而构成一个闭合的螺旋形状, 如图 8.6 所示。因此, Illiac IV 的阵列结构又称为闭合螺旋阵列。这种连接方式既便于一维长向量(多至 64 个元素)的处理, 又便于二维数组运算, 以缩短处理单元之间的路径距离。步距不等于 ± 1 或 ± 8 的任意机间通信可以用软件方法寻找最短路径进行, 其最短距离都不会超过 7 步。例如, 从 PU_{10} 到 PU_{46} 的距离以下列路径为最短:

$$PU_{10} \rightarrow PU_9 \rightarrow PU_8 \rightarrow PU_0 \rightarrow PU_{63} \rightarrow PU_{62} \rightarrow PU_{54} \rightarrow PU_{46}$$

普遍言之, $n \times n$ 个单元组成的阵列中, 任意两个处理单元之间的最短距离不会超过 $(n-1)$ 步。

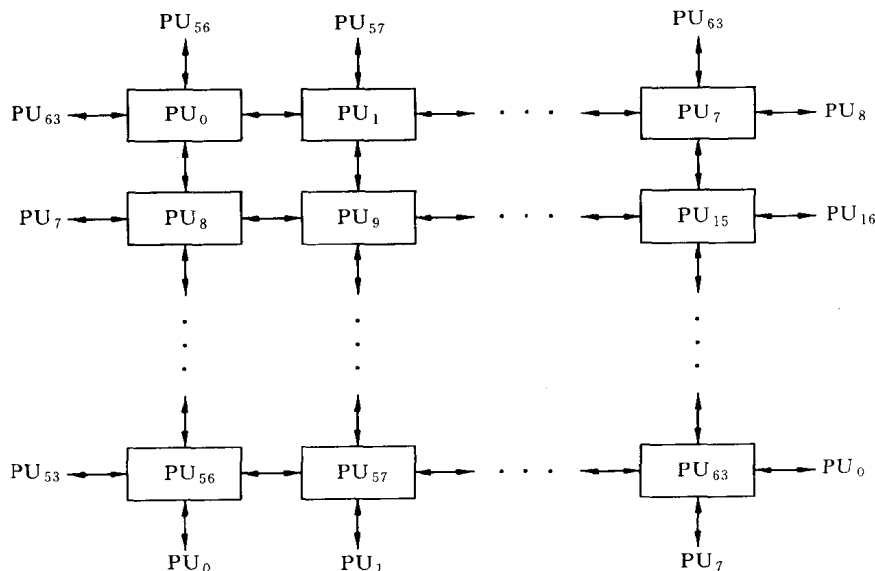


图 8.6 Illiac IV 处理部件的连接

处理单元数组处理的运算部分, 它可对 64 位、32 位和 8 位操作数进行多种算术和逻辑操作, 包括 48 位、24 位或 8 位定点运算。这等于将 64 个处理单元的硬件当作 64 个(64 位)、128 个(32 位)或 512 个(8 位)处理单元发挥作用。并行的加法速度是每秒 10^{10} 次 8 位定点加法或 150×10^6 次 64 位浮点加法。

处理单元的原理框图如图 8.7 所示。它有 6 个可编程序寄存器 RGA、RGB、RGR、RGS、RGX 和 RGM, 以及加/乘算术单元 AU、逻辑单元 LU、移位单元 SU 和地址加法器 ADA 等。RGA 是累加寄存器, 存放第一操作数和操作结果; RGB 是操作数寄存器, 存放加、减、乘、除等二元操作的第二操作数; RGR 是被乘数寄存器兼互连寄存器, 经过东、西、

南、北四个互连路径之一完成处理单元之间的数据直接交往;RGS 是通用寄存器,可被程序用来暂存中间结果。这四个寄存器都是 64 位的。操作数来自以下四个方面:PE 本身的寄存器;阵列存储器;CU 的公共数据总线;PE 的四个近邻。16 位的 RGX 是变址寄存器,它利用地址加法器 ADA 修改指令地址,并将形成的有效地址经过存储器地址寄存器 MAR 输往存储器逻辑部件 MLU。最后是 8 位的模式寄存器 RGM,它的 E 和 E1 位是“活动”标志位,F 和 F1 位保存运算结果出错(上溢、下溢)标志,G、H、I、J 位保存测试结果。RGM 经常处于 CU 的监督之下,一旦出错,就发出 CU 陷阱中断。模式寄存器的活动标志位 E 和 E1 用来控制 RGA、RGS 和阵列存储器的工作,E 还控制 RGX。当 PE 以 32 位字长运算时,E 和 E1 是互相独立的。活动标志位的设立使对 64 个处理单元中的每一个处理单元都可以进行单独控制。只有那些处于活动状态的处理单元才执行单指令流规定的共同操作。RGM 处于程序员的控制之下,可根据其他几个寄存器的内容置为“活动”或“不活动”状态。例如,有一种指令就能在 RGR 的内容大于 RGA 的内容时置该处理单元为“不活动”状态。模式寄存器在阵列处理机中是必不可少的,它对增强阵列处理机的功能和结构灵活性发挥着很大的作用。

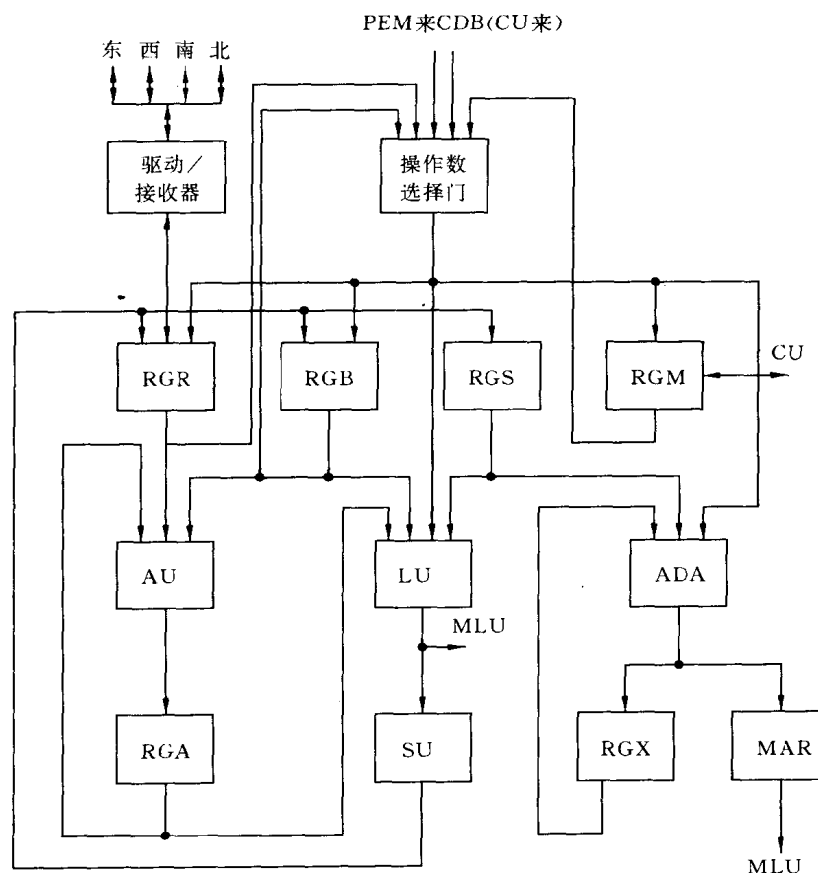


图 8.7 Illiac IV 处理单元原理框图

处理单元存储器 PEM 分属每一个处理单元,各有 $2\,048 \times 64$ 位的存储容量和不大于 350ns 的取数时间。64 个 PEM 联合组成阵列存储器,存放数据和指令。整个阵列存储器可以接受控制器的访问,读出 8 个字的信息块到它的缓冲器中,也可经过 1 024 位的总线与 I/O 开关相连;但是每一个处理单元只能访问自己的存储器。分布在各个处理单元存储器中的公共数据,只能在读至控制器后,再经公共数据总线广播到 64 个处理单元中来。这不但节省了存储空间,而且允许公共数据的存取与其他操作在时间上重叠。这样,阵列存储器就如同一个二维访问存储器:如果把 64 个 PEM 看成列,把每一个 PEM 本身看成行,那么,CU 对它是按列访问,PE 对它是按行访问。

阵列存储器的另一个特点是它的双重变址机构:控制器实现所有处理单元的公共变址,每一个处理单元内部还可以单独变址。最终的操作数有效地址对 PE_i 来说由下式决定:

$$a_i = a + (b) + (c_i)$$

式中, a 是指令地址; (b) 是 CU 中央变址寄存器内容; (c_i) 是 PE_i 局部变址寄存器内容。这种安排增加了各处理单元存储器之间数据分配的灵活性,这对于分别处理矩阵的行和列以及其他维数结构是很有效的。

PE 和 PEM 之间经过存储器逻辑部件 MLU 相连,它包含存储器信息寄存器和有关控制逻辑线路,实现 PEM 分别和 PE、CU 以及 I/O 之间的信息传送。

2. 阵列控制器

阵列控制器 CU 实际上是一台小型控制计算机。它除了对阵列的处理单元实行控制以外,还能利用本身的内部资源执行一整套指令,用以完成标量操作,在时间上与各 PE 的数组操作重叠起来。因此,控制器的功能有以下五个方面:

- (1) 对指令流进行控制和译码,包括执行一整套标量操作指令;
- (2) 向各处理单元发出执行数组操作指令所需的控制信号;
- (3) 产生和向所有处理单元广播公共的地址部分;
- (4) 产生和向所有处理单元广播公共的数据;
- (5) 接收和处理由各 PE(计算出错时)、系统 I/O 操作以及 B 6700 所产生的陷阱中断信号。

Illiac IV 阵列控制器 CU 与处理单元阵列之间的信息联系如图 8.5 所示,一共有以下四条信息通路:

(1) CU 总线 处理单元存储器 PEM 经过 CU 总线把指令和数据送往阵列控制器,以 8 个 64 位字为一信息块。这里指令是指分布存放在阵列存储器中用户程序的指令;而数据可以是处理所需的公共数据,先将它们送到 CU,再利用 CU 的广播功能送到各处理单元。

(2) 公共数据总线 CDB (common data bus) 这是 64 位总线,用作向 64 个处理单元同时广播公共数据的通路。例如,作为公共乘数的常数就不必在 64 个 PEM 中重复存放,可以由 CU 的某一个寄存器送往各处理单元;此外,指令的操作数和地址部分也要经过 CDB 送来。

(3) 模式位线 (mode bit line) 每一个单元都可以经过模式位线把它的模式寄存器 (mode register) 状态送到 CU 中来,送来的信息中也包括该处理单元的“活动”状态位。只

有那些处于“活动”状态的处理单元才执行单指令流所规定的公共操作。从 64 个处理单元送往 CU 的模式位在 CU 的累加寄存器中拼成一个模式字,以便在 CU 内部执行一定的测试指令,对这模式字进行测试,并根据测试结果控制要求的程序转移动作。

(4) 指令控制线 处理单元微操作控制信号和处理单元存储器地址、读/写控制信号都经过约 200 根指令控制线由 CU 送到阵列处理单元 PE 和存储器逻辑部件 MLU 中来。

3. 输入输出系统

Illiac IV 输入/输出系统由磁盘文件系统 DFS、I/O 分系统和 B 6700 组成,而 I/O 分系统又由输入输出开关 IOS、控制描述字控制器 CDC 和输入输出缓冲存储器 BIOM 三个部分组成。

磁盘文件系统 DFS 是两套大容量并行读写磁盘系统及其相应的控制器。每套有 13 台磁盘机,总容量为 10^9 位;每台磁盘机有 128 道,每道一个磁头,并行读写,数据宽度为 256 位,最大传输率为 502×10^6 位/秒;平均等待时间为 19.6 毫秒。如果两个通道同时发送或接收数据,则数据宽度为 512 位,最大传输率可达 10^9 位/秒。

I/O 分系统包括三部分,即输入/输出开关 IOS、控制描述字控制器 CDC 和输入/输出缓冲存储器 BIOM。IOS 的功能有二:一是作为名副其实的开关,用以把 DFS 或可能连上的实时装置转接到阵列存储器,进行大批数据的 I/O 传送;二是作为 DFS 和 PEM 之间的缓冲,以平衡两边不同的数据宽度。CDC 的功能是对阵列控制器的 I/O 请求进行管理。这时,CDC 将使 B 6700 管理计算机中断,由它设法响应 I/O 请求,并通过 CDC 给 CU 送回相应的响应代码,在 CU 中设置好必要的控制状态字。然后,CDC 促使 B 6700 启动 PEM 的加载过程,由 DFS 向 PEM 送入程序和数据。在 PEM 加载完毕后,又由 CDC 向 CU 传送控制信号,使它开始执行 Illiac IV 的程序。至于 BIOM,它处在 DFS 和 B 6700 之间,是为了取得二者之间传送频带的匹配。原来 B 6700 存储器经 CPU 输送数据的频带是 80×10^6 位/秒,而 DFS 输送数据的频带是 500×10^6 位/秒,二者之间相比超过 6 倍之多,因此,必须设立 BIOM 作为 B 6700 和 DFS 之间的缓冲。它把 B 6700 的 48 位字变换为 Illiac IV 的 64 位字,同时以两个字共 128 位的数据宽度输送给 DFS。实际上,BIOM 是用 4 个 PE 存储器做成的,总容量为 8192×64 位。

B 6700 管理计算机的基本组成部分是:单中央处理器(另一 CPU 可选);32 K 字内存(可扩充至 512 K 字);经过多路开关控制的一大批外围设备(包括一台容量为 10^{12} 位的激光外存储器以及 ARPA 网络接口)。B 6700 的作用是管理全部系统资源,完成用户程序的编译或汇编,为 Illiac IV 进行作业调度、存储分配、产生入/出控制描述字送至 CDC、处理中断,以及提供操作系统所具备的其他服务等。

8.3.2 BSP 计算机

BSP 计算机是由美国宝来公司和伊利诺依大学于 1979 年制造的。它是共享存储器结构的 SIMD 计算机的典型代表。BSP 并不是一台独立运行的计算机,它是附属于是系统管理机的一台后端处理机,如图 8.8 所示。

BSP 承担算术运算,系统管理机提供分时服务、数据和程序文件编辑、与远程作业站终端、网络的数据通信、BSP 程序的向量化编译和连接、数据长期存储以及数据库管理等

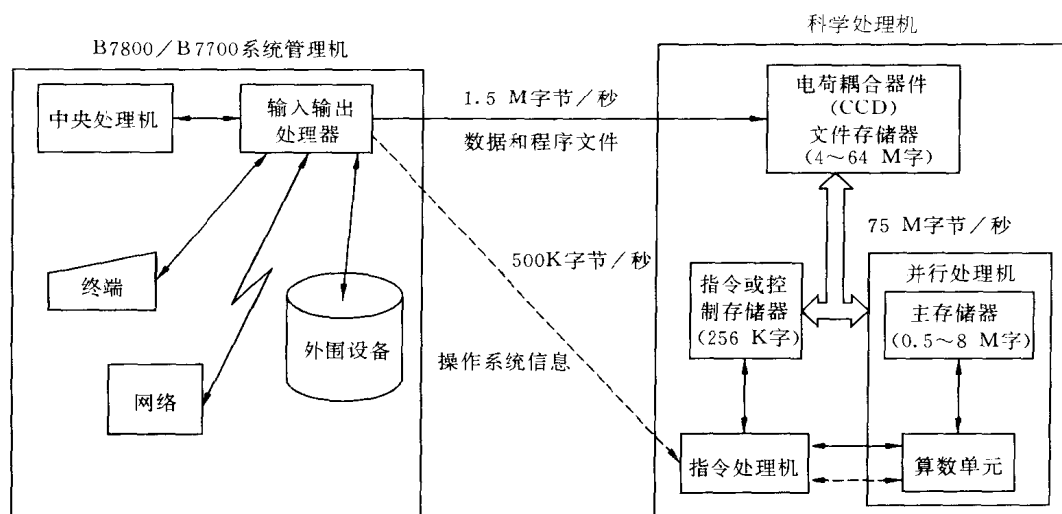


图 8.8 附属于系统管理机的 BSP 计算机

功能。BSP 由控制处理机、并行处理机、文件存储器、并行存储器模块以及对准网络等组成,如图 8.9 所示。

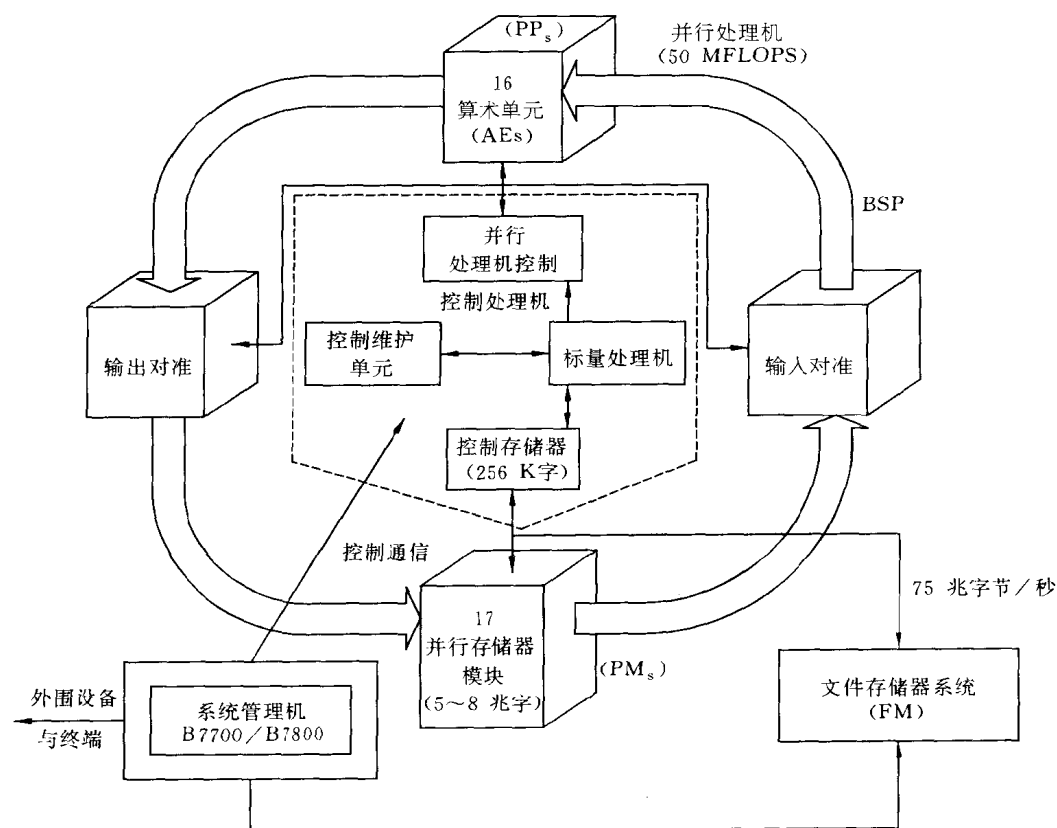


图 8.9 BSP 的功能结构与流水线处理

1. 并行处理机

并行处理机以 160 ns 的时钟周期进行向量计算。所有 16 个算术单元 AE 对不同的数据组(从并行处理机控制器广播来)进行同一种指令操作。大部分的算术运算能在 2 个时钟周期(320 ns)内完成。BSP 的执行速度最高可达 50 MFLOPS。进行向量运算的数据是存在 17 个并行存储器模块中,每个模块的容量可达 512 千字,周期时间为 160 ns。数据在存储器模块和 AE 之间以每秒 100 兆字的速率进行传输。17 个存储器模块的组织形成了一个无冲突访问存储器,它容许对任意长度以及跳距不是 17 倍数的向量实现无冲突存取。

16 个 AE 是以 SIMD 方式在单一微序列控制下同步工作的。在每个 AE 中,只有最原始的操作才采用硬连线方式。控制字的宽度为 100 位。除实现浮点操作以外,AE 还有较强的非数值处理能力。

浮点加、减和乘都能在两个时钟周期内完成。采用两个时钟周期可使存储器频宽与 AE 进行三元操作时的频宽相平衡。所谓三元操作是指由三个操作数产生一个结果的操作。浮点除要用 1 200 ns,是用 Newton-Raphson 迭代算法产生倒数来实现的。在每个 AE 中设有只读存储器,以给出除法和平方根迭代的第一次近似值。浮点字长为 48 位,尾数为 36 位有效值,阶码为 10 位,以 2 为底。数的精度可达到十进制 11 位。AE 在关键部位设置了双字长累加器和双字长寄存器,这就能使双精度运算直接用硬件实现。AE 还可以用软件方法来实现三倍精度的算术运算。可以估算得出来,在 BSP 中用 Fortran 来表达的很大范围的计算问题中,其速度可达 20 到 40 MFLOPS。

2. 控制处理机

控制处理机除了用以控制并行处理机以外,还提供了与系统管理机相连的接口。标量处理机则处理存储在控制存储器中的全部操作系统和用户程序的指令。它以 12 MHz 的时钟频率执行用户程序的串行或标量部分,最高速度可达 1.5 MFLOPS。全部的向量指令以及某些成组的标量指令被送给并行处理机控制器。在经过合格性检查之后,控制器将它们转换为微序列,去控制 16 个 AE 操作。双极型控制存储器的容量为 256 K 字,周期为 160 ns,每个字长 48 位另加 8 位奇偶校验位,提供单错校正双错检测(SECEDD)的能力。控制维护单元是系统管理机与控制处理机其余部分之间的接口,用来进行初始化、监控命令通信和维护。

3. 文件存储器

文件存储器是一个半导体辅助存储器。BSP 的计算任务文件从系统管理机加载到它上面。然后对这些任务进行排队,由控制处理机加以执行。文件存储器是 BSP 直接控制下唯一的外围设备,而其他的外围设备则都由系统管理机来控制。在 BSP 程序执行过程中所产生的暂存文件和输出文件,在将它们送给系统管理机输出给用户之前是存在文件存储器中的。文件存储器的数据传输率较高,大大地缓解了 I/O 受限问题。

4. 对准网络

对准网络包含完全交叉开关以及用来实现数据从一个源广播至几个目的地以及当几个源寻找一个目的地时能分解冲突的硬件。这就需要在算术单元阵列和存储器模块之间具备通用的互连特性。而存储器模块和对准网络的组合功能则提供了并行存储器的无冲

突访问能力。算术单元也利用输出对准网络来实现一些诸如数据压缩和扩展操作以及快速傅里叶变换算法等专用功能。

在 BSP 中,存储器-存储器型的浮点运算是流水进行的。BSP 的流水线组织由五个功能级组成。16 个操作数先从存储器模块中取出,通过输入对准网络送给 AE 进行处理,再将结果经输出对准网络送给存储器模块存储起来。这几级的操作都是重叠进行的,如图 8.9 所示。请注意,在物理上输入对准和输出对准都是在一个实际对准网络中进行的。这里的划分只是对流水级的功能划分而已。除了在 16 个 AE 中显示出的空间并行性以及读取、对准和存储的流水线操作外,AE 中的向量运算还可以同标量处理机中的标量处理重叠起来。这就使得系统既适合于处理长向量和短向量,也能处理单独的标量,系统的功能很强也很灵活。

5. 质数存储系统

BSP 并行存储器由 17 个周期时间为 160 ns 的存储模块组成。由于每个周期存取 16 个字,因此每个字的最大有效存储周期时间为 10 ns。这与算术单元完成浮点加和乘的速率很好地平衡。因为每次运算需要两个变量,算术单元中设有中间寄存器其运算速度为 320 ns/16 次,即 20 ns/次。

由于程序和标量都存放在控制存储器中,因此只有数组存取(包括 I/O)才用到并行存储器。这样,对于三元向量来说,因为在两次算术运算中需要用到三个变量,产生一个结果,共访问存储器 4 次,所以在并行存储器和浮点运算之间的频带保持完全平衡。对于长向量来说,由于中间结果都存在寄存器中,每次运算只需要一个操作数。因此并行存储器有足够的频宽留给输入和输出信息用。

BSP 并行存储器的主要革新是采用了 17 个存储模块。在这之前的巨型机中,普遍采用多个并行存储器模块,但这样的存储系统因访问冲突而使频带严重变窄。例如,若用 16 个存储模块存储 16×16 数组,各行跨模块存储,一列放在一个模块中,则列的存取只能顺序进行。

BSP 提供一种线性向量法来开拓并行性。这一节就是描述实现这一并行性的存储器寻址方法。在 BSP 中,线性向量是反映其并行性的基本参量。它以线性形式将元素映射到主存储器中。线性向量各分量的存储间距为常数 d 。例如,在 Fortran 按列映射的情况下,列分量的间距 $d=1$,行分量的间距 $d=n$,正向对角线分量的间距为 $d=n+1$ 。在 BSP 中处理线性向量既利用了空间并行性也利用了时间并行性。

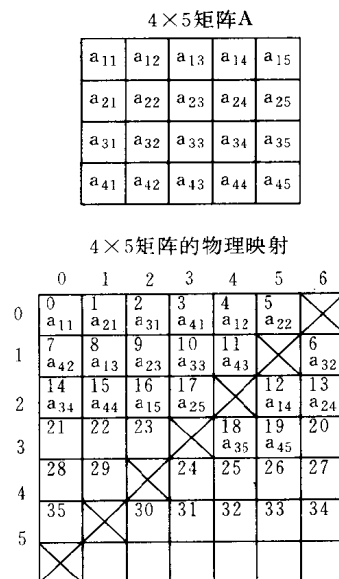
BSP 的一个独特的性能就是它的存储系统可以无冲突访问,在每个存储周期内送给每个 PE 一个有用的操作数。存储器中向量元素的间距不一定为 1。因而 DO 循环可以有非单位增量,或者程序可以访问矩阵的行、列和对角线而无需付出额外的代价。巨型机的设计者或对使用存储器作严格的访问限制,或采用昂贵的快速存储器件,用很宽的存储器频带来获得一定程度无冲突访问的功能。

在 BSP 中保证无冲突访问的硬件技术包括:质数个存储器端口;存储器端口和 AE 间的完全交叉开关,以及特殊的存储器地址生成机构,它为具体的地址模式计算出合适的地址。这里的地址模式是指正统的串行计算机所用的那一种模式。这就是说,每个较高的存储地址是指存储器中的“下一个”字。并行存储器采用这种模式能与当前程序设计语言

的所有结构完全兼容。特别是 Fortran 的 EQUIVALENCE、COMMON 以及数组参数传送都可以用常规计算机上一样的方法来实现。

现在来讨论一台含 N 个 AE 和 M 个存储器模块的类 BSP 机的情况,存储器模块号 μ 用来指定哪一个模块内存储与线性地址 a 有关的数据元素。模块号可以用 $\mu = a(\bmod M)$ 计算出来。在所指定的存储器模块内的地址偏移量 i 可以用 $i = \left\lfloor \frac{a}{N} \right\rfloor$ 求得。

图 8.10 画出一个 4×5 矩阵按列映射到一台串行机存储器中的情况。为使说明简单起见,假定此假想机中的 $N = 6, M = 7$ (BSP 中的 $N = 16, M = 17$)。模块号和偏移量的计算结果都列在图中。模块号以一定周期(等于 AE 数目)重复出现,增量为 1。偏移量与同一模块重复出现的次数相对应,它在一个周期(等于存储器模块的数目)是不会重复的。



例如,若 $M=7, N=6$, 则 4×5 数组映射为

数组元素	a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}	a_{15}	a_{25}	a_{35}	a_{45}
线性地址 a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
存储器模块号 μ	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5
模块内偏移量 i	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3

图 8.10 矩阵元素线性化向量的 BSP 存储器映射

只要 AE 数目小于或等于存储器模块的数目,则求得的偏移值序列总可使不同的存储器模块与每一个 AE 相连接。这样,每个 AE 接收或发送的数据都是唯一的。在 6 个 AE、7 个存储器模块的系统中为 4×5 数组产生的存储模式如图 8.10 所示。现将数组第二行元素所对应的存储器模块和偏移值的计算解释如下。起始地址为 1,跳距为 $d=4$ 。因此可求得下列的模块号:

$$\begin{aligned} \mu &= 1(\bmod 7), 5(\bmod 7), 9(\bmod 7), 13(\bmod 7), 17(\bmod 7), \\ &= 1, \quad 5, \quad 2, \quad 6, \quad 3, \end{aligned}$$

相应地可求出每个存储模块内的偏移值:

$$i = \left[\frac{1}{6} \right], \left[\frac{5}{6} \right], \left[\frac{9}{6} \right], \left[\frac{13}{6} \right], \left[\frac{17}{6} \right]$$

$$= 0, \quad 0, \quad 1, \quad 2, \quad 2,$$

决定对 $N=16$ 个 AE 采用 $M=17$ 个存储器模块将给最普通的数组划分提供无冲突访问的能力。由于一个周期内只有一个存储器模块没有用到,因此存储器频带的冗余度仍很小。很清楚,对一维数组的任何算术数列指标模式,除每一个第 17 个元素以外,可以对它无冲突进行访问。对二维数组来说,若将错开距离定为 4,则对行、列、正向对角线、反向对角线以及其他普通划分都能无冲突访问。这一方法可以用直截了当的方式推广到高阶维数的情况。

之所以产生没有用的存储单元,是因为 AE 数比存储器模块数少一个的缘故。不过,所举的例子并不是实用的情况。对于含 16 个 AE 和 17 个存储器模块的真正的 BSP 来说,被 16 除比被 17 除的情况处理起来要简单和快得多。为了达到一个给定的有用的规模,需要付出一定的代价设置一些额外的存储器。上面给出的几个式子造成了以 AE 为中心的有利局面。只要同一组式子总是作用于一开始就从 I/O 进来的数据,则存储模式对用户来说完全是看不见的。由于硬件总是按同一原则工作,因此这种情况也适用于程序的转储。

假如数组元素的地址相隔是存储器模块数的整数倍时,则冲突一定会发生。在这种情况下,所有要访问的值都处于同一个存储器模块中。对 BSP 来说,即应该避免跳距为 17、34、51 等的情况。实际上,51 确是有问题的跳距。因为这正好是列长度为 50 的数组的正向对角线元素存储地址的跳距。如果在 BSP 中发生冲突,其运算仍可正确进行,但速度下降到正常速度的 $1/16$ 。而系统却能记录冲突以及它们对总的运行时间影响的情况,以便在这种影响太大时使程序员采取一定的改进措施。

BSP 可以对下列四类操作进行并行计算:

- (1) 16 个算术单元实现并行运算;
- (2) 存储器的读取和存储以及存储器和算术单元间的数据传输;
- (3) 在并行处理机控制器中的变址值、向量长度和循环控制计算;
- (4) 线性向量操作描述字在标量处理机中的生成。

8.3.3 CM-2 计算机

Connection Machine 的 CM-2 是一台细粒度的 SIMD 计算机,它由数千个位片 PE 组成。它的峰值处理速度超过 10 Gflops。它的系统结构如图 8.11 所示。所有程序从前端开始执行,当需要并行数据操作时,发送微指令到后端处理阵列。定序器(sequencer)分解这些微指令并且把它们广播给阵列中的所有数据处理器(data processor)。前端机和处理阵列之间有三条交换数据计算结果的通路:广播总线(broadcasting)、全局组合总线(global combining)和标量存储器总线(scalar memory bus)。广播是通过广播总线把数据或指令同时传送到所有数据处理器。前端机通过全局组合总线对来自各处理器的数据进行求和、最大值、逻辑或等运算。前端机每次通过标量总线从与数据处理器相连的存储器

读取 32 位数据、或者每次将 32 位数据写入与数据处理器相连的存储器。VAX 和 Symbolics 机都可以用作前端机和主机。

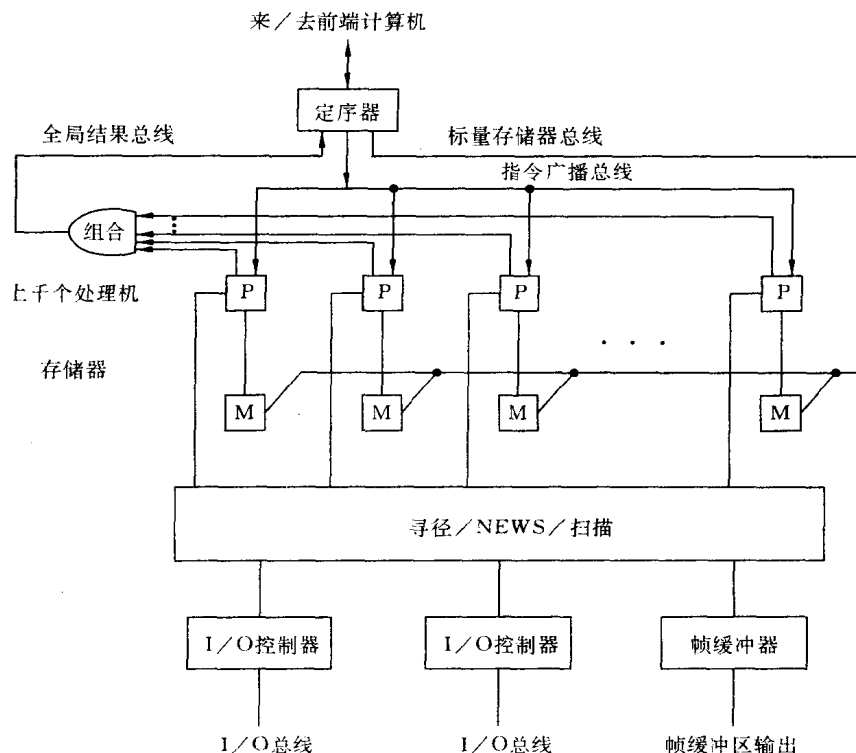


图 8.11 CM-2 的系统结构

1. 处理阵列

CM-2 是一台数据并行计算的后端机。处理阵列包含 4 K 到 64 K 个位片数据处理器（或 PE），所有数据处理器都由定序器控制，如图 8.12 所示。

定序器对来自前端机的微指令进行译码，然后把毫微指令广播到阵列中各个处理器。所有处理器可以同时访问它们的存储器，它们以锁步方式执行广播来的指令。

处理器之间通过寻径、NEWS 网格 (NEWS grid) 或扫描机构 (scanning mechanism) 相互交换数据。这些网络也与 I/O 接口相连。称为数据穹 (data vault) 的大容量存储器子系统与 I/O 相连，它可存储多达 60 G 字节的数据。

CM-2 处理器芯片、存储器芯片和浮点芯片如图 8.12 所示。每个处理器结点包括 32 个位片数据处理器、一个可选的浮点加速器和处理器之间通信的接口。每个数据处理器用 3 个输入和 2 个输出的位片 ALU、有关锁存器和存储器接口实现。ALU 可以执行位串全加操作和布尔逻辑操作。

每个结点有一对处理器芯片，它们共享一组存储器芯片。每个处理芯片有 16 个处理器。称为 Pairs 的并行指令系统包括许多毫微指令，它们用于存储器的装入和存储、算术和逻辑运算、寻径器控制、NEWS 网格控制、超立方体接口控制、浮点运算、I/O 和诊断操作。

每个处理器芯片的存储器数据路径宽度是 22 位(16 位数据和 6 位 ECC)。18 位存储器地址允许 32 个处理器共享 $2^{18}=256\text{ K}$ 个存储器字(512 K 字节数据)。浮点芯片一次执行 32 位的操作。中间计算结果可以存入存储器供后续运算使用。整数算术运算直接由处理器以位串方式执行。

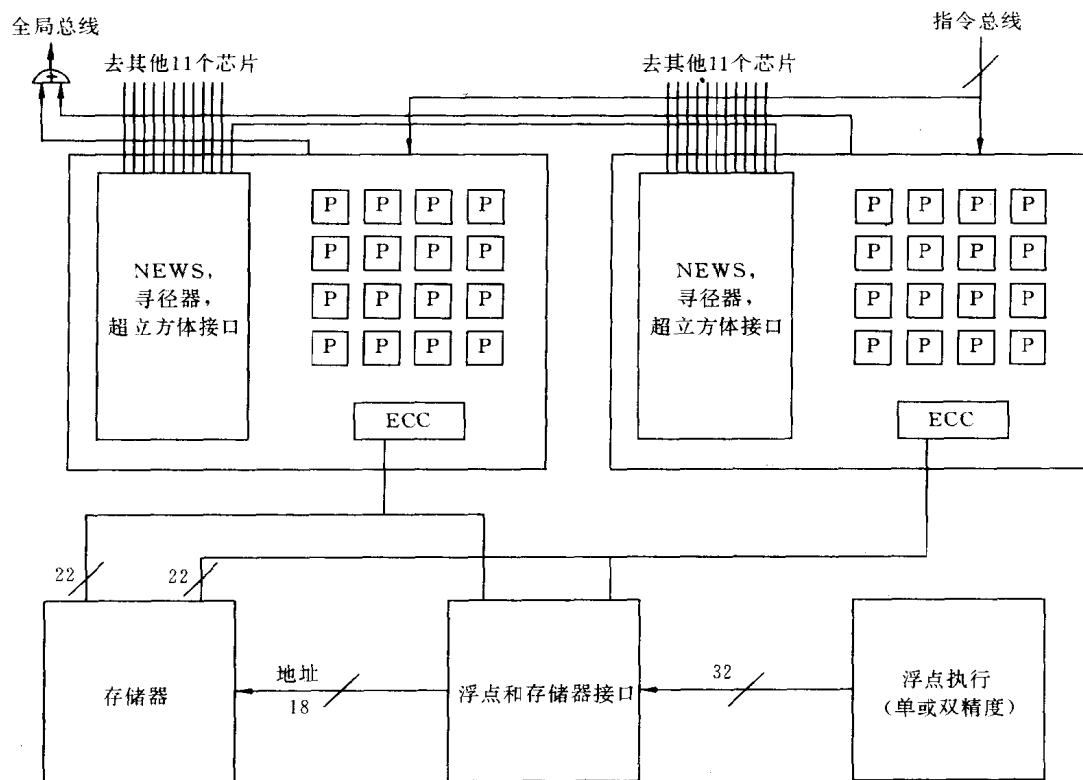


图 8.12 由两个处理器芯片、一组存储器和浮点芯片组成的 CM-2 处理结点

2. 寻径器、NEWS 网络和扫描机构

(1) 寻径器 每个处理器芯片包含一个用于处理器之间数据寻径的专门硬件。把所有处理器芯片上的寻径器结点用线连在一起形成一个布尔 n -立方体。在 CM-2 最大配置时,所有处理器芯片上共有 4 096 个寻径器结点,连成一个 12 维的超立方体。

每个寻径器结点与 12 个其他寻径器结点相连接,其中包括它的对偶结点在内(图 8.12)。属于同一结点上的 16 个处理器在发送消息给 12-立方体另一顶点上的任一处理器时具有相同的能力。下面说明这种消息传递的概念。

例如 12-立方体的每个顶点上的处理器编号为 0 到 15。4 096 个顶点的超立方体的寻径器编号为 0 到 4095。第 7 号寻径器结点上的第 5 号处理器在整个系统中的编号为 117, 因为 $16 \times 7 + 5 = 117$ 。

假设处理器 117 要发送一个消息给处理器 361。处理器 361 位于第 22 号结点的第 9 号处理器($16 \times 22 + 9 = 361$)。由于寻径器结点 $7 = (00000000000111)_2$, 寻径器结点 $22 = (00000001010)_2$, 这两个二进制编号在维 0 和维 4 不同。

这一消息一定通过维 0 和维 4 到达它的目的结点。如果不存在争用超立方体连线现象,那么消息从寻径器结点 7 开始,首先沿维 0 方向传到寻径器结点 $6 = (00000000110)_2$,然后再沿维 4 方向传到寻径器结点 22。如果寻径器点 7 号的另一消息正在使用维 0 方向的连线,那么为了避免发生通道冲突,消息传送的路径可以改变为先沿维 4 方向传到寻径器结点 $23 = (000000010111)_2$,然后再沿维 0 方向到达最终的目的结点。

(2) NEWS 网格 每个处理器芯片中的 16 个物理处理器可以排列成 $8 \times 2, 1 \times 16, 4 \times 4, 4 \times 2 \times 2$ 或 $2 \times 2 \times 2 \times 2$ 等形式的网格。规定每个物理处理器有 64 个虚拟处理器。可以想象这 64 个虚拟处理器在芯片中排列成 8×8 网格。

NEWS 网格是建立在各种不同配置网格的每个处理器都有东、南、西、北四个邻居这一事实的基础上的。而且,可以选择一部分超立方体的连线把 2^{12} 个结点连成一个任意形状的二维网格, 64×64 是一种可能的网格形状。

把每个结点的内部网格与全局网格结合在一起,就可以把处理器排列成任意维任何形状的 NEWS 网格。这些灵活的处理单元互连方式使得数据非常有效地沿着根据应用要求而建立的专门网格进行传递。

(3) 扫描机构 除了通过超立方体寻径器可以动态地重构 NEWS 网格外,CM-2 还有专门的硬件支持对整个 NEWS 网格的扫描或传播。这些都是很有效的并行操作,在整个阵列中进行快速的数据组合和传播。

对 NEWS 网格扫描是把通信和计算结合在一起。可以沿某一维方向同时对网格的每一行扫描,求出该行的部分和、找出最大值或最小值、进行按位或、与、异或计算。扫描操作可以扩展到对整个阵列的所有元素进行。

传播能将一个数据传送给其他芯片上的处理器。一位二进制数只用 75 步就可以沿着超立方体连线从一个芯片送到所有其他芯片。为了便于访问,各种形式的扫描和传播操作已经设计在 Pairs 指令中了。

3. 输入输出系统

Connection Machine 不但强调计算的大规模并行性,还强调计算结果的可视化。2 到 16 条高速 I/O 通道用于数据和/或图象 I/O 操作。连接到 I/O 通道的外围设备包括数据穹、CM-HIPPI 系统、CM-IOP 系统和 VME 总线接口控制器,如图 8.11 所示。数据穹是基于磁盘的海量存储系统,用来存放程序文件和大数据库。

CM-2 已经用于解决几乎所有 MPP 所面临的具有重大挑战性的应用问题。特别是 Connection Machine 系列已经用于借助相关反馈技术的文档检索、基于记忆的推理,如用在医疗诊断系统 QUACK 中模拟诊断疾病以及处理工作量很大的自然语言等。CM-2 的其他应用还包括 SPICE 的 VLSI 电路分析和布线、计算流体动力学、信号/图象/视觉处理和集成、神经网络模拟和连接模型、动态规划,上下文无关文法分析、射线追踪图以及计算几何等问题。

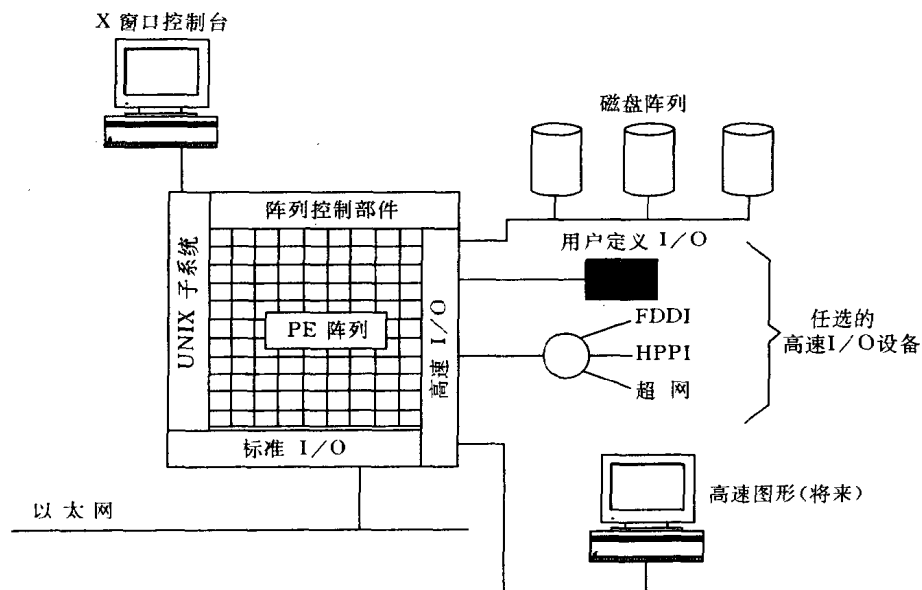
8.3.4 MasPar MP-1 系统

MasPar MP-1 是一台中粒度 SIMD 计算机。它的系统结构如图 8.13(a)所示。

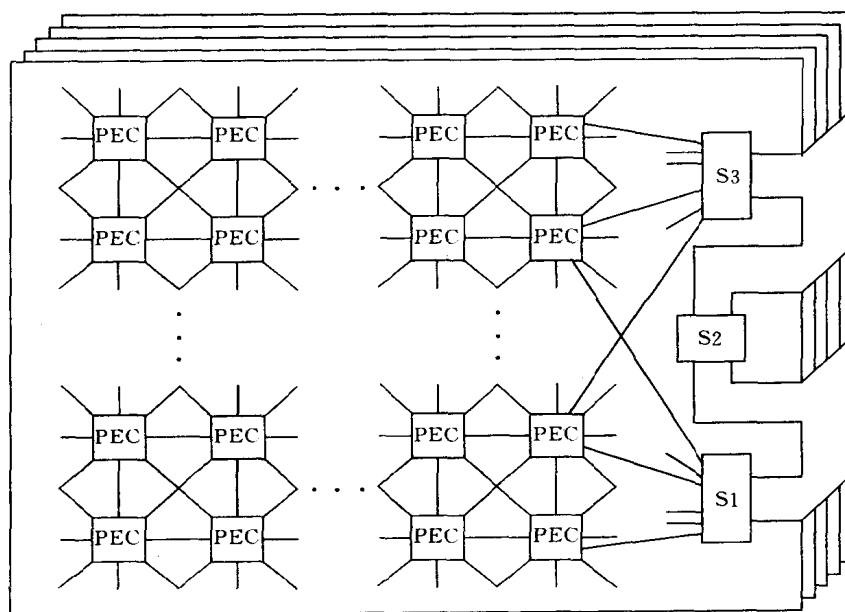
MP-1 系统结构由 4 个子系统组成:PE 阵列、阵列控制部件(ACU)、具有标准 I/O 的 UNIX 子系统和高速 I/O 子系统。UNIX 系统负责传统的串行处理。高速 I/O 与 PE 阵列

协同工作负责大规模并行计算。

MP-1 系列有 1024、4096、最多可达 16 384 个处理器的各种配置。当配置为 16 K 个处理器时,32 位 RISC 整数操作的峰值性能是 26 000 MIPS。系统浮点运算峰值性能单精度操作为 1.5 Gflops,双精度操作为 650 Mflops。



(a) MP-1 系统框图



(b) PE 群阵列

图 8.13 MP-1 系统结构

1. PE 阵列

每块处理器板有 1 024 个 PE 和有关的存储器,排列成 64 个 PE 群(PEC),每个群有 16 个 PE。图 8.13(b)给出了每块处理器板上 PEC 之间的连接形式。每个 PEC 芯片通过 X-Net 网格形网络和一个在图 8.13(b)中的标为 S1、S2 和 S3 的全局多级交叉开关寻径器网络与 8 个相邻的 PEC 芯片相连。

每个 PE 群(图 8.14(a))由 16 个 PE 和 16 个处理器和存储器(PMEM)组成。16 个 PE 在逻辑上排列成一个 4×4 的 X-Net 两维网格形互连阵列。群内的 16 个 PE 共享多级交叉开关寻径器的一个访问端口。处理机之间通过下述三种机制进行通信:

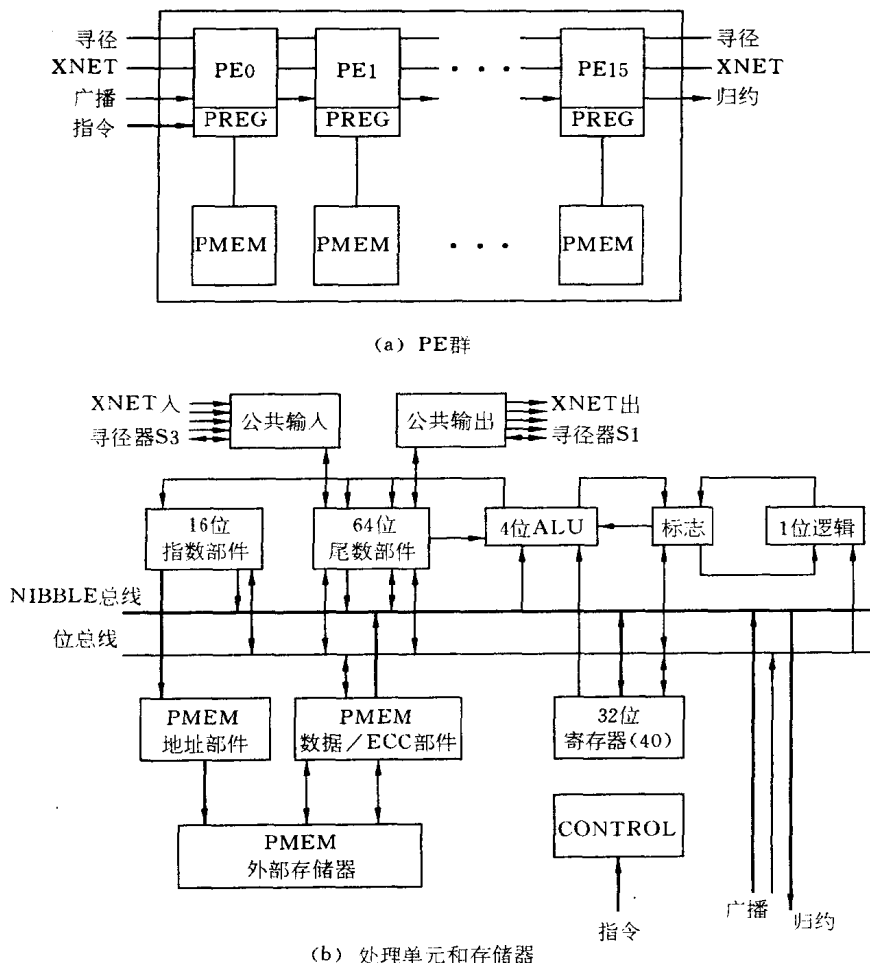


图 8.14 MP-1 处理单元和存储器的设计

(1) ACU-PE 阵列通信

第一种机制支持 ACU 把指令/数据同时广播到阵列中的所有 PE,并且对并行数据进行全局归约以便从阵列中收回标量值。

(2) X-Net 近邻通信

X-Net 将每个 PE 与它的两维网格上的 8 个相邻 PE 直接相连。每个 PE 在它的对角线上有 4 个连接,构成一个与 BLITZEN X 网络网络类似的 X 模式,每个 X 交叉点上的三状态结点允许与 8 个相邻 PE 中的任何一个进行通信,每个 PE 只使用 4 条线。

PE 阵列四周的连接线绕过来形成 2 维的环网。环网结构对称,并且容易实现几种重要的矩阵算法。还可以在 X-Net 上用 2 步来模拟一维环。当 MP-1 配置最大时,X-Net 的总通信频宽为 18 G 字节/秒。

(3) 全局交叉开关寻径器通信

多级交叉开关互连网络能实现所有 PE 之间的全局通信,并且可作为 MP-1 I/O 系统的基础。三级寻径器实现了一个 1024×1024 交叉开关的功能。每块处理机板有 3 个寻径器芯片。

每个 PE 群共享一个与 S1 级寻径器相连的源端口以及与 S3 级寻径器相连的目的端口。所建立的连接是从源 PE 经过 S1、S2 和 S3 然后到达目的 PE。MP-1 全配置时有 1024 个 PE 群,因此,每级有 1024 个寻径器端口。每个寻径器同时支持多达 1024 个连接,总频宽为 1.3 G 字节/秒。

2. 处理单元和存储器

PE 设计主要是数据通路的逻辑设计,PE 中没有取指和译码逻辑。设计框图如图 8.14(b)所示。整数和浮点数计算在每个基于寄存器的 RISC 结构的 PE 中执行。取数和存数指令将数据在 PEM 和寄存器组之间传送。

每个 PE 有 40 个供程序员使用的 32 位寄存器和 8 个供系统使用的 32 位寄存器。寄存器可按位和按字节寻址。每个 PE 有一个 4 位整数 ALU、一个 1 位逻辑部件、一个 64 位尾数部件、一个 16 位指数部件和一个标志部件。NIBBLE 总线的宽度是 4 位。BIT 总线的宽是 1 位。PEM 可直接或间接寻址。存储器最大频宽为 12 G 字节/秒。

每个 PE 的大多数数据传送都在 NIBBLE 总线和 BIT 总线上进行。每个微步可以同时激活 PE 内的不同的功能部件。换句话说,可以同时执行整数、布尔数和浮点数操作,每个 PE 的运行时钟频率不高。与 CM-2 类似,它通过大规模并行性来提高速度。

3. 阵列控制部件

ACU 是一台 14 MIPS 的标量 RISC 处理机,使用请求调页的指令存储器。ACU 负责取指令并进行译码以及计算地址和标量数据、发送控制信号到 PE 阵列、控制 PE 阵列的状态。

与 CM-2 中的定序器类似,ACU 采用微程序编码,对 PE 阵列实现水平控制。大多数 ACU 标量指令的执行时间为一个 70ns 时钟周期。整个 ACU 做在一块 PC 板上。

一个称为存储器机器的功能部件与 ACU 并行工作。存储器机器执行 PE 阵列的装入和存储操作,而 ACU 则将算术、逻辑和寻径指令广播到各个 PE 并行执行。

4. 并行磁盘阵列

值得一提的另一个特点是在 MP-1 中实现大规模并行 I/O 系统结构。PE 阵列(图 8.13(a))通过高速 I/O 子系统与并行磁盘阵列通信。I/O 子系统主要用 1.3 G 字节/秒的全局寻径网络来实现。

磁盘阵列格式化后的容量高达 17.3 G 字节,磁盘的持续 I/O 速率为 9 兆字节/秒。为

了支持数据并行计算,为了提供透明的文件系统和多级容错能力,设置并行磁盘阵列是十分必要的。

8.4 SIMD 计算机的应用

8.4.1 数值应用问题的特征

大型数值应用问题一般都是高度结构化的计算问题,一些大程序需要 $10^{12} \sim 10^{15}$ 次浮点运算才能达到要求的精度。为了解决这类大型问题,必须提高系统结构的并行性,并且要充分研究这类问题的特点。

设计者首先必须了解那些大型题目的特殊要求,然后才能设计出满足这类问题的极端特殊需要的方案。设计者甚至还必须熟悉其他有关领域的要求,努力选择一个能够同时满足几个领域要求的方案。只有这样他们的方案才是可行的。因为如果一个系统结构能为几个不同领域服务,那么设计费用、软件开发以及其他的开发费用就可由多个用户共同分担,从而大大降低每个计算机用户的费用。

计算机设计人员必须仔细研究那些能扩大机器用户数目的设计方案,对各种用户要求的最高性能采取折衷的办法以便适应各种用户的需要。例如,假设一台计算机进行向量运算时能达到最高性能,而对于某些只要求标量运算的用户来说,这又有什么用呢?在这种情况下设计人员必须提高该机器处理标量运算的性能,这样做对于仅仅要求标量运算和仅仅要求向量运算的用户都会有很大影响。这两类用户都可能由于不能充分发挥设计能力而蒙受损失。

以上在向量计算机上进行标量计算的例子恰巧可以说明高速执行标量运算的重要性。一个高效的系统结构设计方案,即使在绝大多数情况下是为了用于向量运算,也应该具有较高的标量运算的性能。一般系统结构的设计方案对标量运算与向量运算的性能要求是一致的,而其他方面的要求就不那么明确了。例如输入/输出处理、互连网络、数据通路以及系统结构所支持的算法等都没有统一的标准。

在这些方面怎样设计才能满足不同用户的需要呢?本节试图探讨数值应用本身的问题,揭示真正对计算机系统结构产生重大影响的问题的特征。

1. 大型数值问题的分类

本节所叙述的大型计算是可以用来模拟物理过程的数字处理技术。这类问题的共同特点是:

- (a) 由于数值的动态变化范围相当大,所以在很大程度上依赖于浮点算术运算。
- (b) 计算模型用时空上的离散点来模拟物理上连续的空间和时间的函数。
- (c) 算法的设计者可以选择不同的离散模型表达式,这样在确定问题的大小时具有很大的灵活性。进行蒙特卡罗模拟(随机模拟)时,设计者可选择网格点的数目、时间步的大小以及跟踪轨迹点的数目。这些数值必须根据系统结构的特点适当地加以选择。
- (d) 算法设计者往往以牺牲运行时间来换取运算精度。一般来讲答案的精确度要求越高,为达到这一精度进行计算时选取的点的数目也就愈多,在执行计算时所需要的机器周期也越长。

以上这些特点说明即使在某些专用的系统结构中,算法设计者仍然可以修改算法使其适应于该系统结构。例如在 64 台处理机的 Illiac IV 系统结构中,很容易把网格点分配给各台处理机,因而各处理机可并行地对各网格点进行计算。然而对于 65 个网格点的计算任务,Illiac IV 系统结构存在严重的缺陷。因为第一次可以将 65 个网格点中的 64 个分配给 64 台处理机,这样每台处理机处理一个点。当它们处理完后,剩下的一个点还需分配给 64 台处理机中的一台去执行,而其余 63 台处理机则处于空闲状态。所以,实际上处理 65 个网格点比处理 64 个点的效率明显地降低。

在 Illiac IV 上运行的程序其数据网格点的数目很少完全适合于它的结构,然而执行效率又与数据网格点的数目密切相关。实际上,一个典型题目的模拟点的数目是可以由算法的设计者确定的,而不是问题本身所固有的。既然在一定范围内可灵活地决定问题的规模,所以算法设计者可以调整问题的规模,使得它们能在 Illiac IV 结构上高效率地运行。

不幸的是高性能机器的算法依赖于机器的系统结构。算法的设计者在确定了适合 Illiac IV 的网格点的数目之后,为了在其他类型的机器上实现该算法,就不得不重新设计算法。

目前高级语言和优化编译技术的发展,使得在普通机器上运行的 FORTRAN 语言程序,可以通过优化编译器的处理转换成可在高度并行的机器上运行的程序,从而有效地利用并行机的优越性。最好的编译器已经超过了具有编写并行计算机程序一般经验的程序员的水平,但经验丰富的程序员通常比编译器工作得更出色。

遗憾的是使用高性能的机器时,为了充分利用处理器的资源,原有的大量程序需要进一步加以改进,以便提高运行效率,这类工作是值得的。通过人工编码取代自动代码生成,可以将执行速度提高一倍,因而计算时间将缩短几小时或几天。

在近期内,算法的设计者在设计算法时应尽量利用机器系统结构的潜力。这种对系统结构的依赖性加重了软件开发工作的负担。而长远目标是利用编译技术来完成这项工作,把为通用结构的机器写的算法转化为适合于高性能的特殊结构机器的形式。

所以计算机系统结构设计者对自己所设计的计算机的应用对象的特点需要进行全面的了解。无论是科学计算还是数值计算,事务处理还是数据处理,或是非数值的问题都应如此。

(1) 连续模型

哈辛诺在 1986 年将物理计算模型分成两类:

- 1) 连续模型;
- 2) 离散模型(粒子模型)。

连续模型中计算的时间和空间是连续变化的物理量,典型的参数如电荷密度、温度和压力等。这些物理量是一定范围内的平均值。粒子模型则将世界看成是由离散的粒子组成的,典型的参数如速度、力和动量等,这些物理量反映单个粒子的当前状态。

例如,考虑等离子体的温度这个概念,通常将等离子体看作是内部温度按物理规律连续变化的一个实体。而离散观点将等离子体内各粒子看作一个单独实体,而不直接体现温度概念。然而,等离子体的温度是各点速度和密度的直接函数,可以通过取许多点的平均值计算出来。

数字计算机是个离散设备,因而所有的数字计算都以离散形式进行,无论它们原先是否连续。但模拟计算机可以模拟连续问题,而不需要将其转换成离散的形式。比如“风洞实验”,“风洞”就是一台可以计算流体流动的模拟机。并行计算机上运行的一些大型题目在价格低廉的模拟机上运行时速度非常低,因而数字计算机的应用引起人们的高度重视。例如,目前可模拟各种速度和其他我们感兴趣的问题如“风洞”的计算机,其价格超过十台超级数字计算机的价格。

连续模型和离散模型的主要区别如下:

连续模型符合偏微分方程。按离散形式处理时,方程式中所有变量的变化都是其近邻变量的函数,远程变量没有直接的影响。先通过作用于近邻变量,近邻变量再作用于随后的近邻变量,由此向前非直接地作用于整个媒质,也就是通过局部作用而将作用传送到整个媒质。

离散模型(粒子模型)允许粒子受远程粒子的影响。任何粒子在任何时刻都与模型中的其他粒子有关。

连续模型的好处是模型中的每个点都类似于一台独立的自动计算机,每个点可以通过考察其邻近的点来确定自己的状态。每个点根据它目前的状态和其邻近点的状态,进行计算并修改其状态。每个点的计算都是独立地、并行地进行的。所以当观测热对流效应、流体流现象或其他连续物理过程时,可将动态过程看成是每个连续点根据局部和邻近数据进行少量的计算。如果这种连续模型正确地反映过程的特征,那么用此模型进行计算的高度并行计算机的特征也就很清楚了。

(2) 粒子模型

如果连续模型对于并行计算很适用的话,那么粒子模型可能就不太适用了。这是由远程作用的特点造成的。每个粒子必须检查所有粒子的状态后才能确定它究竟会发生什么变化。当模型中的粒子数增加时,每个粒子的运算量将相应增加,而总计算量的增加比粒子数目的线性增长要快得多。这与连续模型截然不同,在连续模型中每个点的计算和模型大小毫无关系。但由于这两个模型可用来描述同一个物理过程,所以我们应该将它们结合起来考虑。

例如,万有引力是远程作用的物理现象中一个极好的例子。由于大量核子的相互作用,物质在宇宙中产生或湮灭,一个物体的质量发生变化时,它受到地球的万有引力也随之变化。

“牛顿的苹果”从树上掉下来正是远程作用的结果。苹果与宇宙中所有其他物体的相互作用产生引力,使苹果产生一个加速度。离散模型也可用来分析苹果受力的情况,宇宙中所有其他物体对苹果都有作用力,这些力的合力使苹果落到地上。

相互作用的引力与距离有着密切的关系,因为引力与物体的质量成正比,与距离的平方成反比,所以很远的物体可以忽略不计,除非它们的质量很大。对地球上的大多数物体来说,由于地球的质量非常大,所以只考虑地球产生的引力就可以了。但也有例外,如潮汐现象还需要考虑太阳和月球的引力。当研究宇宙飞船的轨迹时,将太阳系中的行星、较大的卫星以及太阳考虑进去就可以了,至于太阳系中的小物体和太阳系外的物体都可以忽略。

在粒子物理学中也有类似的情况,只是其中大部分粒子都要考虑进去。另外,由于力和距离的平方成反比,所以离得非常远的粒子的影响可以不予考虑,除非它们的体积很大或者密度很大,以至总的作用力很大。考虑远程粒子的作用时,若围绕着离散粒子的球体其半径的变化较小时,球体体积的变化与半径的平方成正比,即: $dv=4\pi^2dr$

如果空间中粒子密度是均匀的,并且作用力随着半径的平方的增大而减小,那么在球心处受到的力的总和是一个与球体半径无关的常量。实际上球心处受到的力会随着半径增大而减小,逐渐趋向于零。

如果粒子密度随着体积的增大而减小,那么粒子总的受力也会随着远程粒子距离的增大而减小。所以我们在这里学到的第一个原则就是远程作用实际上是有界限的,因而我们可以根据模型特点将注意力放在附近部分粒子的范围之内。

第二个原则是关于连续模型中的远程效应。即使连续模型中所有的计算都是局部的,显然远程条件最终也会影响每个点。例如,在一个热交换模型中,当金属的表面温度升高时,温度将通过金属传递,最终使整个物体温度升高。如果热量通过其他表面散发,那又是另一回事了。连续模型中的近邻计算相当于热传导物理模型中的热量从一个点传到另一个点,直到每个点都同样地受到影响为止。

如果我们用一个相应的离散模型来模拟同样的过程,物体内的每个粒子都必须检查其他粒子,包括那些与受热表面有关的粒子。如果每个粒子直接计算受热表面对它的作用,并且适当地考虑固体内部的热消耗,那么每个粒子可以迅速改变其原来的状态,但这并不是一个简单的计算。

连续观点产生的计算模型并行性高且局部性好,但这种模型传递一个作用需要一定的延迟时间。离散(粒子)观点产生一个粒子之间都是直接作用的模型,这种模型处理的传递速度很快,但计算变得非常复杂,需要计算的作用数目很大。

在某些环境下,连续模型所需要的基本算术操作比离散模型少。例如,几个活动粒子的作用的传递在连续模型中是通过相邻粒子传递给每个离散粒子的。几个不同粒子的作用到达某个中间的粒子点时,所有的作用就合成为一个,然后传递这个合成的作用而不是传递几个单独的作用。在连续模型中,一个点之所以仅需考虑其近邻点的作用而无需考虑所有点的作用也正是这个原因。所有的作用最终合并成一个并通过其相邻点作用到每一点。因此用不着检查所有作用,这些作用量通常是很大的,连续模型将作用合并起来,以减少每个点对作用的计算量。

当粒子的数目很大时,作用数目以粒子数的平方增长。另一方面,连续模型要求的计算量与连续模型的点数、相邻点数以及最长传播路径长度的乘积成比例。因为路径长度通常以点数的平方根或立方根增长,所以连续模型的操作量会比离散模型小得多。通常连续模型中的点数与离散模型中的粒子数毫不相关,所以这两个模型不能直接比较。

2. 适合于连续模型的 SIMD 系统结构

连续模型对于并行计算尤其具有吸引力,它为解决许多实际问题提供了一种相当简洁的方法。早期的并行计算机主要解决这类问题,这一方面是由于并行性是这种模型固有的特性,另一方面是由于一些重要的大型问题都可在连续结构中得以解决。

(1) 阵列结构

这一节我们主要讨论为什么要设计阵列处理机结构,并且说明这种系统结构怎样有效地解决除多用途之外的各种设计问题。这类系统结构的极特殊的结构限制了它仅适用于属于连续模型的问题。即使是连续模型问题,这种基本系统结构有时仍不能满足所有要求。

作为连续模型的例子,我们用泊松方程描述某个区域内的电位与该区域内的电荷密度之间的函数关系,其二维的形式如下:

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = -Q(x,y) \quad (8.1)$$

其中 $V(x,y)$ 是点 (x,y) 的电位, $Q(x,y)$ 是该点的电量。方程在某个区域内的解与边界条件有关,边界条件可以用许多方法表示。对这个具体例子,我们假定要在区域 $0 \leq x \leq 1, 0 \leq y \leq 1$ 内求解这一方程,我们将得到这个区域边界上的值 $V(x,y)$ 。

连续方程不能直接在数字计算机上求解,首先要把它转化成能进行数字化处理的离散形式。为此,把连续区域表示成若干个离散的点,如图 8.15 所示的网格形式。方块表示网格中的一个结点,方块中的整数表示坐标。网格的交叉处 (i,j) 存放着 $V(i,j)$ 和 $C(i,j)$ 的值,下标 i,j 的范围为 $0 \sim N-1$,所以 x 和 y 的值为 $x = i/N$ 和 $y = j/N$ 。

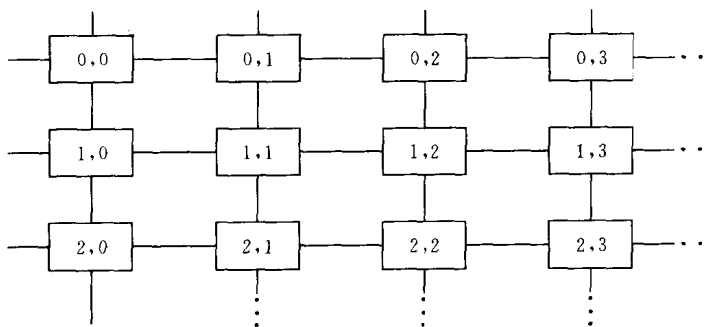


图 8.15 连续空间的网格表示

如果区域中的点充分靠近,就可以得到非常近似的电位值。用离散方式描述问题的精确度完全依赖于网格的间隔。当然,点的数目随着间隔的缩小以平方的倍数增加,随着间隔的变小计算时间将会变得很长。用户必须在模型的精度和计算的费用之间取得平衡。随着计算速度的提高,用户可以增加网格的数目,以便提高解决问题的精度。

连续物理空间变换成离散空间后,我们可以把连续方程变换成对应的离散方程。方程(8.1)需要知道二阶导数,这可通过求一阶导数的离散近似值得到。因此,我们先考虑解决一维问题的连续的一阶导数,如方程

$$dV(x)/dx = -C(x) \quad (8.2)$$

根据导数的经典定义,导数可表示为如下的离散形式:

$$dV(x)/dx = (V_{i+1} - V_i)/(1/N) \quad (8.3)$$

当 $N+1$ 个点等距离地分布在 0 到 1 的空间时,分母 $1/N$ 是网格的间距。方程(8.3)表示了 x 轴上点 i 和点 $i+1$ 正中间的那个点的导数值。点 i 的导数值可近似地由下式给出:

$$dV(i/N)/dx = (V_{i+\frac{1}{2}} - V_{i-\frac{1}{2}})/(1/N) \quad (8.4)$$

式(8.4)中格点 $i + 1/2$ 和 $i - 1/2$ 是假想的。如果要用真实的格点,可以在等式(8.4)中使用 $i + 1$ 和 $i - 1$, 同时把分母改为网格间距的两倍。其实这样做并没有必要,因为在推导二阶导数时假想的格点会被消去。

现在我们计算一维空间中二阶导数的离散近似公式:

$$\frac{d^2V(x)}{dx^2} = \frac{d}{dx} \left(\frac{dV(x)}{dx} \right) = \frac{(V_{i+1} - V_i) - (V_i - V_{i-1})}{\left(\frac{1}{N}\right)} = \frac{(V_{i+1} + V_{i-1} - 2V_i)}{\left(\frac{1}{N}\right)} \quad (8.5)$$

把(8.5)式代入(8.1)式,得到的离散公式

$$(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1})/4 = V_{i,j} - C_{i,j} \quad (8.6)$$

$$0 < i < N, 0 < j < N$$

因此,网格中每一点的电位是相邻四点的平均值与该点电量值之和。方程(8.6)是一个求解网格内部各点电位的线性方程组,包含 $(N-2)^2$ 个未知量 $V(x,y)$ 。通过解这个方程组,我们可以得到一个与原连续系统非常接近的近似解。

可以用标准的线性方程组解法求解方程(8.6)。但这个方程组很特殊,这是个稀疏的高度结构化的方程组。为了提高效率,我们应该尽量利用方程组的特殊形式。

在众多方法中,有一种对并行系统结构最有吸引力。这种方法隐含地而不是直接地求方程的解,最基本的思想是使用如下形式的迭代:

$$V_{i,j}^{(t+1)} = \frac{V_{i+1,j}^{(t)} + V_{i-1,j}^{(t)} + V_{i,j+1}^{(t)} + V_{i,j-1}^{(t)}}{4} + C_{i,j} \quad (8.7)$$

式(8.7)中上标 t 表示迭代次数。这样,每一点的值用其四邻的平均值与该点电量的和来修改。当各点的值不再变化时迭代终止,得到最后结果。

一种可用于求解(8.7)这种形式的方程组的计算机系统结构是阵列处理机。每个格点有一台处理机,每台处理机与其四邻直接相连,通过这种连接,处理机可得到方程(8.7)所必须的值。由于所有的处理机执行同样的迭代,只要一个指令流就可以控制所有处理机。专门由一台控制处理机播送指令,网格结点中的各台执行计算的处理机接收并执行指令。

程序 8.1 是在这种结构上求解泊松方程的一段程序。假设每台处理机都有一个存储器,一组通用寄存器,一个连接东南西北四邻的路径寄存器。

程序 8.1 的指令格式是: **操作码,目的,源**。其中操作码表示要执行的操作的类型,目的是存放结果的地址,源是一个操作数的地址。如果操作数有两个,如 ADD 和 MULTIPLY,则第二个操作数的地址由目的给出。指令 MOVEREG[1],MEM[v]把本地存储器 V 单元的内容传送给寄存器 1。相反,指令 MOVEMEM[V],REG[1]把寄存器 1 的内容送到存储器的 V 单元。

当处理机之间进行数据传送时,每个数据首先由指令 MOVEROUTE, MEM[V]从存储器传到路径寄存器或由指令 MOVEROUTE, REG[1]从通用寄存器传到路径寄存器。处理机之间的数据传送由指令 ROUTE 完成,如指令 ROUTENORTH 表示把每个处

理机的路径寄存器中的数据传送给它的北邻的路径寄存器。执行指令 ROUTENORTH 后,路径寄存器中立即获得了其南邻传送过来的数据。指令 ROUTENORTH 既是 SENDTONORTH 指令,又是 RECEIVEFROMSOUTH 指令。

程序 8.1 阵列处理机上求解泊松问题的主迭代

```

    LOOP;LOAD REG[1],MEM[V] ; 本地电位值送 REG[1]
    MOVEROUTE,REG[1]        ; 准备传送数据
    ROUTENORTH               ; 数据向北邻传送
    MOVEREG[2],ROUTE        ; 从南邻接收数据
    MOVEROUTE,REG[1]
    ROUTEEAST                ; 数据向东邻传送
    ADDREG[2],ROUTE          ; 加上两邻送来的数据
    MOVEROUTE,REG[1]
    ROUTESOUTH               ; 数据向南邻传送
    ADDREG[2],ROUTE          ; 加上北邻送来的数据
    MOVEROUTE,REG[1]
    ROUTEWEST                ; 数据向西邻传送
    ADDREG[2],ROUTE          ; 加上东邻送来的数据
    DIVREG[2],4              ; 求平均值
    ADDREG[2],MEM[C]         ; 人加上本地电量值
    MOVEMEM[V],REG[2]       ; REG[2]中放 V 的新值,REG[1]中放 V 的老值下面加上
                                ; 检测收敛的指令
                                ; .....

```

根据上述约定,程序 8.1 可用来求解(8.6)那种方程。程序中没有用到下标,因为格点 (i,j) 正好在处理机 (i,j) 中处理。如果网格点数超过处理机数目,则可将整个网格分割成更小的区域,每个小区域的大小与物理网格的大小相同,这样便可处理较大的网格了。在这种情况下,每一小区域分配给整个物理网格,每台处理机需解方程若干次,每次对应不同的区域,相邻区域的计算串行地进行。为区分不同的区域,计算区域 K 时,程序中 MEM[V]变为 MEM[V[K]]。

阵列处理机系统结构对于连续模型是很理想的,但处理其他模型提出的要求缺乏灵活性。其实连续模型的许多计算也会有这类要求。例如,当要解的问题是空间坐标的函数或时间的函数时,网格的大小可能需要动态地改变。而如前所述,这种系统结构只适于处理大小统一的矩形网格。

一些计算问题可分成两个或多个区域。每个区域又有不同的计算,而上述系统结构不能并行地进行不同的计算,所以,不同的计算必须串行地在部分处理机上进行。

从根本上说,这种系统结构应用领域很狭窄,所以用户很少,这样每个用户的费用也就很高。此外,用户为一个新的系统结构开发软件的工作量太大。

新的系统结构需要新的工具如编译程序、调试程序以及操作系统等等。开发这些工具需要花费大量时间。而没有这些工具,用户就只好在低效率的环境下编制程序。因此,要使一种系统结构对用户具有吸引力,除了它本身的高性能外,还必须配备良好的软件工具

并且价格合理。

Illiac IV 的运行实践证明了它对于设计时所针对的那些问题十分有效。它与 64 台处理机相配合,速度大约是单台 Illiac IV 处理机的 10 到 50 倍。但是后来更快的元器件产生了更有竞争力的机器。64 台处理机的 Illiac IV 变得不如那些并行度较低但速度很快的机器有吸引力。当 Illiac IV 被淘汰时,用户开始转向那些非专用机,这些机器有许多应用软件以及新型的速度更快的硬件,且大多数程序的性能都并不比 Illiac IV 差。

回顾连续模型机器发展的历史,从中可以得出如下一些重要结论:

① ILLIAC 系统结构处理连续模型的效率非常高,它对处理连续模型的高速商品化机器的系统结构设计有很大的影响。

② 仅适合于解决连续模型问题而不适合于解决其他模型问题的系统结构用户必然很少,与拥有大量用户的系统结构相比,用户需增加费用,需要自己编写专用软件。

③ 高性能串行处理机的速度每三至四年就翻一番,因此 10 年之内可以提高速度 10 倍,所以加速比为 10 的并行计算机系统结构的有效期大约为 10 年。

④ 如果软件支持工具的研制拖延了时间,那么这种系统结构的有效期还将缩短。如果像优化编译程序这类关键工具没有研制出来,这种系统结构给用户带来的好处可能会全部丧失。

⑤ 为特殊的用途研制一种最合适的系统结构从用户的利益来说并不是非常必要。不管哪种系统结构都应有一定的适用范围以及一定数量的用户。

(2) 多立方体结构

前面我们已经介绍了由连续模型产生的高度并行的近邻通信的计算方法。大范围的并行计算以锁步方式同步地在所有处理机上同时进行。为了利用这些特点,系统结构设计者研制了一些机器,其互连方式反映了连续模型的近邻结构。我们已经简略地讨论了 Illiac IV 计算机,它的每个结点与其四个近邻互连在一起,这种二维的网格结构看来很适合数学网格的计算。

塞茨曾经设计过一种称作多立方体的系统结构,它的 64 台处理机组成一个逻辑上的六维立方体。每一台处理机和立方体的六个方向上的近邻处理机直接相连。

为了具体地说明这种连接方式,我们给每一台处理机一个六位二进制的标号。例如处理机(0,0,0,0,0,0),用它来表示该处理机在六维空间的坐标。那么与处理机(0,0,0,0,0,0)相邻的是那些标号中只有一个坐标不同的处理机,如(1,0,0,0,0,0),(0,1,0,0,0,0),(0,0,1,0,0,0),……,和(0,0,0,0,0,1)

多立方体中的“维”数反映了一种处理机之间的互连关系,是从算法方面考虑的,与物理上的时空维数毫无关系。尽管连续模型是能够在多立方体上高效率地运行的一类主要的算法,然而多立方体也并不是只适合于连续模型的计算。

多立方体中的处理机相互独立,每台处理机能够运行各自的程序。在一台模型机中,每台处理机实际上是一台由 8086 处理机芯片和 8087 浮点处理芯片构成的微处理机。多立方体并不要求所有结点同时执行相同的指令。相反,各个结点可以独立地运行各自的指令,相邻处理机之间可以通过消息进行通信。

多立方体的系统结构与 Illiac IV 和其他连续型系统结构的差异主要有以下两点:

① 多立方体的互连方式反映了不同算法的需要,而 Illiac IV 的互连方式只反映一种特殊的网格几何结构。

② 多立方体的处理机的独立性允许处理机系统同时进行不同的计算。而 Illiac IV 系统中的处理机必须同时执行同一条指令(或者处于空闲状态)。

由 64 台处理机组成的 Illiac IV 的性能相当不错,在当时它是一个成功的系统,但只适用于某类问题。研究人员将许多不同类型的问题在 Illiac IV 上运行,结果表明如果能把问题变成适合于 Illiac IV 上处理的形式,那么就能达到相当高的速度。他们找到了解决这类问题的新技术,同时也发现 Illiac IV 的近邻连接结构确实非常好,但也并不是最佳的互连结构。Illiac IV 系统结构的最大缺陷是不能同时运行不同的程序。然而这有一个权衡的问题,因为独立操作有同步问题,同步需要一定的额外开销,另外,由于竞争共享资源会使性能下降。

3. 数据流需求

在这一节,我们将说明大规模数字计算中所固有的数据流的类型。尽管这些算法只是例子,我们认为其数据流的需求是有代表性的,高性能的机器必须支持这类数据流才能达到高效率,多立方体的设计恰好满足这些需求。这部分地说明了为什么它采取这种互连方式。

为了说明这种基本信息流,图 8.16 给出了一个有 N 个输入端和 N 个输出端的计算装置。我们假定各输入端同时输入,经过一段延迟后,所有输出端同时产生输出信号。我们感兴趣的是一类对计算装置内部数据流要求最严格的函数。

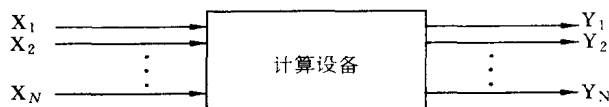


图 8.16 全信息函数

这类函数的特点是每个输入信号将影响所有输出信号,每个输出信号受到所有输入信号的影响,我们称这类函数为全信息函数。任何一个能计算这种全信息函数的计算装置有一个互连网络,从它的每一个输入到每一个输出之间都要有一条内部路径。我们可以测出输入端和输出端两两间的最短路径,并且其中最长的路径是最坏情况下计算时间的上界。至少有一组输入要沿着那条最长的路径(或沿着一条较长的路径)传送数据,同时至少有一个输出要依赖于这个传送得最慢的数据。

符合粒子模型的计算往往是全信息函数。例如,考虑粒子相互作用模型中粒子的运动情况,每个粒子都受到系统中其他粒子的作用力。反过来,每个粒子又给系统中其他粒子一个作用力,把关于每个粒子的信息传送给系统中其他的粒子,由此可以计算出每个粒子作为时间函数的位置和速度。由于要求每个输入均作用于每个输出,计算装置体现了“远距离作用”这一思想。

图 8.17 说明排序也是一个全信息问题。图 8.17 是一个能够把输入数据按升序进行排序并输出的装置。图 8.17(a)的输入端是一组已经排好序的数据,输出端的顺序没有发生变化。对于这组特殊的输入数据,这组装置的内部结构也许非常简单。

然而,在图 8.17(b)中,我们改变了第一个输入端的数据,发现每个输出都发生了变

化,因此第一个输入端一定与每个输出端相连。显然,每个输入都同样地影响着每个输出。改变第二个输入端同样会看到类似于图 8.17(b)的情况,也就是说,单个输入端的变化,可以使所有输出值发生改变,可见排序的确是一个全信息函数。图 8.17(a)的第一个输入端的值改变后,导致图(a)的输出变成图(b)的输出。每个输入都影响每个输出。

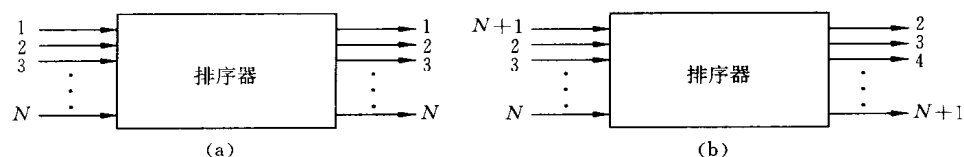


图 8.17 排序是一个全信息函数

熟悉信号分析技术的读者也许会想到傅里叶变换也是一个全信息函数。假设计算装置的输入数据是 N 个时间样本,还假设此计算装置计算这个信号的傅里叶变换。它的输出是信号的频谱,即第 i 个输出端的输出值是基频的第 i 个谐波的振幅。频谱依赖于所有时间样本。如果任何一个时间样本发生变化,那么每个频率的振幅都可能发生变化。

我们有时忽略了在连续模型中进行计算时也可能是全信息函数。因为连续模型看起来好像很适合于近邻连接,所以连续模型中的计算有时是全信息函数会使人感到很奇怪。但是,对于要求全信息交换的计算,近邻连接仅仅是信息传递的通路,距离遥远的两点一定会相互影响。

泊松方程是连续模型的典型例子。图 8.18 是物理空间中的一个网格,网格中某些点带有电荷。网格中每点的电位值取决于所有其他各点的电荷的多少。前面介绍的计算电位值的并行算法只使用了近邻的信息,但必须重复进行计算直到答案收敛为止。

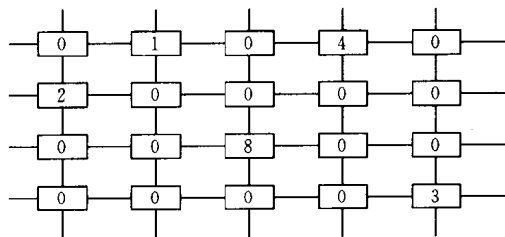


图 8.18 解决连续模型泊松方程问题的网格

实际上每次重复计算把网格中每个结点的电位的影响沿着网格向外传播远一些。如果网格的每维有 N 个结点,那么要使每个结点都能影响其他所有结点,至少需要重复计算 N 次。如果采用图 8.16 所示的计算装置来解决连续问题,互连方式就不仅限于近邻互连方式,那么计算时间不再随网格线性增加,计算速度提高了。

这种信息流的思想在并行计算机设计中很有用。图 8.17 的排序网络模型代表了所有排序网络的特性。假定我们使用一种逻辑装置作为排序网络的基本构造模块,它的扇入和扇出系数是一个很小的常数,例如 2。从图 8.19 的树形结构中很容易看出每个装置最多能影响在它下面的 K 层共 2^K 个装置。由此可知,一个用于 N 项数据排序的网络至少有

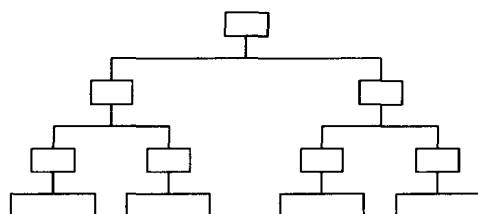


图 8.19 扇出系数为 2 的逻辑网络

$\log_2 N$ 个逻辑层。

如果扇入和扇出系数大于 2, 而仍小于某一常数, 那么层数仍将以 $\lg N$ 增长, 只是对数的底等于扇入和扇出系数。消去对数的唯一方法是扇入和扇出系数无限大。

在程序 8.1 中, 我们只考虑近邻之间的数据流。这段程序并不是一个完整的迭代程序, 因为它计算出近邻的平均值后就不再迭代。实际上应该根据测试结果判断是否收敛, 如果没有收敛还需要重复进行迭代, 直到网络中所有结点的值都收敛时才能结束。但测试收敛需要得到网络中所有结点的信息, 而近邻互连这种结构没有供测试收敛用的直接通路。

8.4.2 算法举例

下面举几个算法的例子。

1. 有限差分问题

有限差分方法是求解场方程的一种有效方法。它把一个有规则的网格覆盖在整个场域上, 用网格点上变量值写出差分方程组以代替场方程来进行计算。在解决物理问题时, 如果将描述平面场的拉普拉斯方程

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (8.8)$$

中的二阶偏导数表示为差分形式

$$\left. \begin{aligned} \frac{\partial^2 U}{\partial x^2} &= \frac{U(x+h, y) - 2U(x, y) + U(x-h, y)}{h^2} \\ \frac{\partial^2 U}{\partial y^2} &= \frac{U(x, y+h) - 2U(x, y) + U(x, y-h)}{h^2} \end{aligned} \right\} \quad (8.9)$$

并代入原方程, 则可得有限差分计算公式

$$U(x, y) = \frac{U(x+h, y) + U(x, y+h) + U(x-h, y) + U(x, y-h)}{4} \quad (8.10)$$

式中, (x, y) 为平面直角坐标, h 为网格间距。

Illiac IV 的阵列结构特别适用于计算这种在网格上定义的有限差分函数。这是因为, 根据式(8.10), 任一网格点 (x, y) 上的函数值可由其四周邻近点的函数值计算出来, 这一要求正好反映了阵列处理机每一处理单元与其四个近邻连接的性质。实际计算时, 应利用张弛法进行。每一网格点上的函数值用求其四邻平均值的方法计算, 经多次迭代, 逐次逼

近其最终的平均值。网格边缘的函数值是已知的,由场域的边界条件决定;而对于内部各点的函数值,开始时可选择为零,然后根据式(8.10)多次迭代求 $U(x,y)$ 值,直至连续二次迭代所求值的差小于规定误差为止(已知迭代过程是收敛的)。

Illiac IV 在计算时,是把内部网格点分配给各个处理单元的。因此,上述计算过程可以并行地完成,从而可几十倍地提高处理速度。由于实际问题中所遇到的内部网格点数目往往是很大的,因此需要将其分成许多子网格,然后才能在 Illiac IV 上求解。

2. 矩阵加

在阵列处理机上解决矩阵加法问题是最简单的一维情形。假定两个 8×8 的矩阵 A 和 B 相加,则只需把 A 和 B 居于相似位置的一对分量存放在同一 PEM 内,且令 A 的分量在全部 64 个 PEM 中存放的单元地址均为 a , B 的分量单元地址码均为 $a+1$,而 $C = A + B$ 的结果分量均放在地址码为 $a+2$ 的单元内。如图 8.20 所示。于是,只需用下列三条 Illiac IV 的汇编指令就可一次实现矩阵相加。

```
LDA    ALPHA      ; 全部(a)由 PEM 送 PE 的累加器 RGA
ADRN   ALPHA+1    ; 全部(a+1)与(RGA)进行浮点规舍加,结果送 PGA
STA    ALPHA+2    ; 全部(RGA)由 PE 送 PEM 的 a+2 单元。
```

由于全部 64 个处理单元并行操作,故速度可提高为顺序处理的 64 倍。

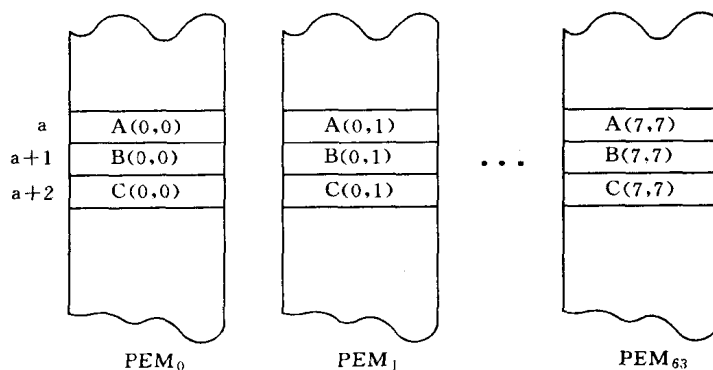


图 8.20 矩阵相加存储器分配举例

3. 矩阵乘

由于矩阵乘是二维数组运算,故它比矩阵加要复杂一些。设 A 、 B 和 C 为三个 8×8 的二维矩阵。若给定 A 和 B ,则 $C = A * B$ 的 64 个分量可利用下列公式计算。

$$c_{ij} = \sum_{k=0}^7 a_{ik} b_{kj}, \quad 0 \leq i, j \leq 7 \quad (8.11)$$

如果在 SIMD 计算机上求解这个问题,则可执行下列 FORTRAN 程序

```
DO 10 I=0,7
  C(I,J)=0
  DO 20 K=0,7
    20 C(I,J)=C(I,J)+A(I,K)*B(K,J)
  10 CONTINUE
```


类似的程序在 SISD 计算机上要用 K, I, J 三重循环才能完成, 每重循环执行 8 次, 共需 512 次加乘时间(不考虑其他循环控制指令所需时间)。而在 SIMD 计算机上执行上面的程序时, 如果利用 8 个处理部件并行计算, 则 J 循环只需一次即可完成, I, K 循环照旧。这样, 便可使速度提高到 8 倍, 即缩短为 64 次加乘时间。程序流程图如图 8.21 所示。

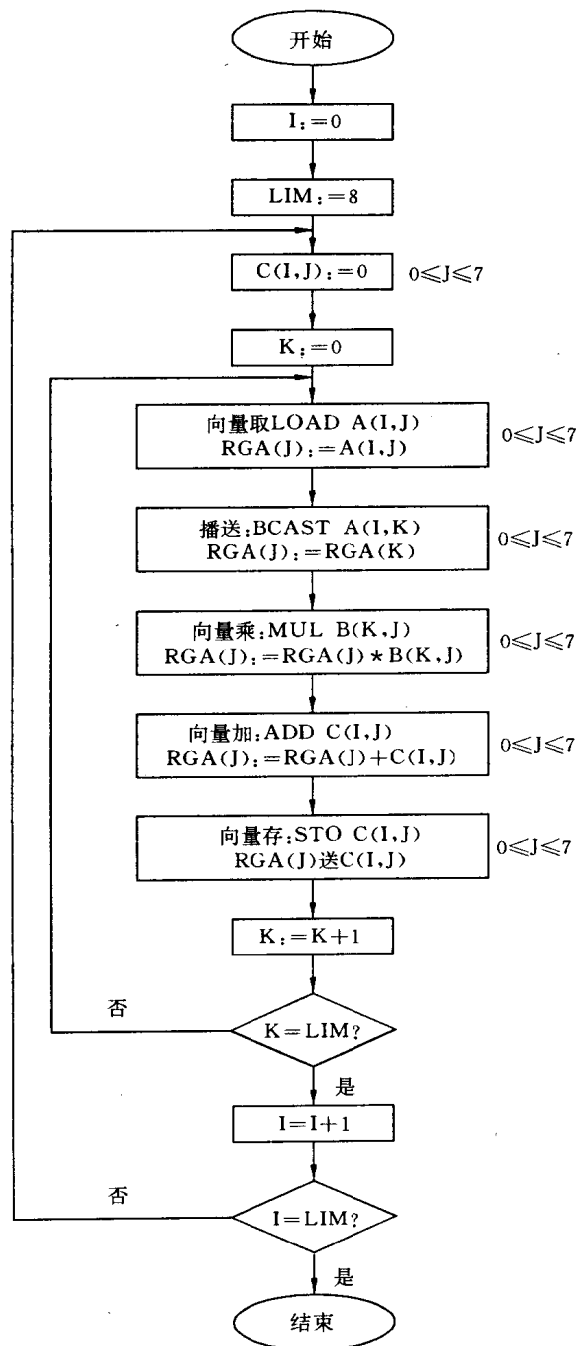


图 8.21 矩阵乘程序执行流程图

从表面上看,这个程序在每个处理部件内部的执行过程和传统的 SISD 计算机是一样的,但是由于 8 个处理部件执行同一指令流,并行操作,故实际解决问题的方式是不同的。第一,控制器执行的指令,表面上是标量指令,但实际上等效于向量指令(如向量取、向量存、向量加、向量乘等)。第二,执行这个程序对于 A、B、C 向量在处理单元存储器中的分布方案提出了一定的要求,如图 8.22 所示。作乘法时,操作数 $B(K,J)$ 都从本处理部件的 PEM 中读取;但被乘数 $A(I,K)$ 对所有处理部件 $0 \leq J \leq 7$ 都是一样的,它一般都不在本处理部件存储器内。因此,要利用阵列处理机的“广播”功能,把一次循环的公共系数 $A(I,K)$ 取出后送到控制器,再广播到全部 8 个处理单元的 RGA 中去。

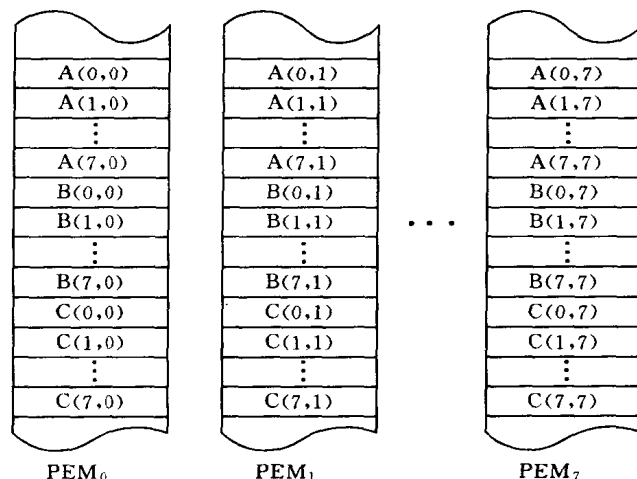


图 8.22 矩阵乘存储器分配举例

如果把 Illiac IV 的 64 个处理单元全部利用起来并行运算,即把 K 循环的运算也改为并行,则可进一步提高速度。要做到这一点,需要重新在阵列存储器中恰当分配数据,还要能使 8 个中间积 $A(I,K) * B(K,J)$, $0 \leq K \leq 7$, 并行相加。这就是下面要讨论的累加和问题。

4. 累加和

这是一个将 N 个数的顺序相加过程变为并行相加过程的问题,要用到处理单元中的活动标志位。只有处于活跃状态的处理单元,才能执行相应的操作。为叙述方便,取 $N=8$ 。假定有 8 个数 $A(I)$, $0 \leq I \leq 7$, 顺序累加。在 SIMD 计算机上可写成下列 FORTRAN 程序:

```
C(-1) = 0
DO 10 I=0,7
10 C(I)=C(I-1)+A(I)
```

这是一个串行程序。在 SISD 计算机上,它要用 8 次加法时间。如果在并行处理机上,采用成对递归相加的算法,则只需 $\log_2 8 = 3$ 次加法时间就够了。首先,原始数据 $A(I)$, $0 \leq I \leq 7$, 存放在 8 个 PEM 的 a 单元中,然后按照下面的步骤求累加和:

第一步 置全部 PE 为活动状态

- 第二步 全部 $A(I), 0 \leq I \leq 7$, 从 PE 的 a 单元读到相应 PE 的 RGA 中;
- 第三步 令 $K=0$;
- 第四步 全部 PE 的 (RGA) 转送到 RGR;
- 第五步 全部 PE 的 (RGR) 经过互连网络向右传送 2^k 步距;
- 第六步 $j=2^k-1$;
- 第七步 置 PE_0 至 PE_j 为不活动状态;
- 第八步 处于活动状态的 PE 执行 $(RGA) := (RGA) + (RGR)$ 操作;
- 第九步 $k := k+1$;
- 第十步 若 $k < 3$, 则转回第四步, 否则继续往下执行;
- 第十一步 置全部 PE 为活动状态;
- 第十二步 全部 PE 的 (RGA) 存入相应 PEM 的 $a+1$ 单元中。

图 8.23 为其计算过程示意图。其中, 画有影线的处理单元表示被屏蔽; 框中数字表示各次循环结果。为简单起见, $A(0) \sim A(7)$ 等在表和图中分别以 $0 \sim 7$ 等数字代表; 且第五步中 PE 的 (RGR) 在右移时超出 PE_7 的内容就不必表示了。这是因为, 若右移步距为 $2^k \pmod{N}$, 则它们应移入 PE_0 至 PE_j , 但既然这些 PE 在第七步将要置为不活动状态, 则无论它们的 RGR 接受什么内容, 都是不起作用的。

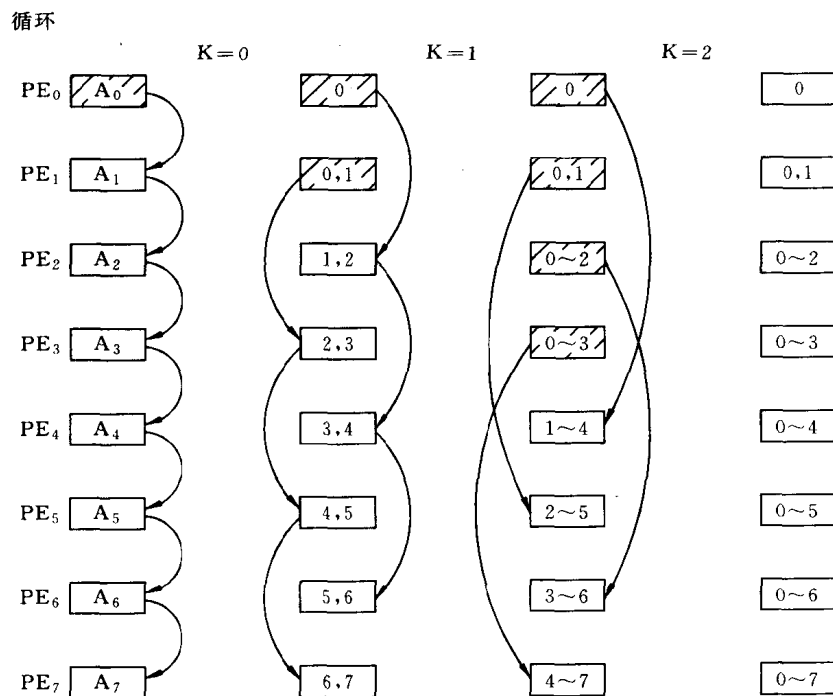


图 8.23 并行处理机上累加和计算过程示意图

这个例子表明: 虽然经过变换, 原来的串行程序也能在 Illiac IV 上执行, 但由于屏蔽了一部分处理单元, 降低了它们的利用率, 所以速度提高的倍数不等于处理单元的个数 N , 而只是 $N/\log_2 N$ 。

5. 递归技术

给出了信息必须如何流动的总限制后,现在来讨论某些专门用于解决数值问题的并行技术。假定有如下并行计算:

$$\begin{aligned} X_i &= a_i + X_{i-1}, \quad i > 0 \\ X_0 &= a_0 \end{aligned} \quad (8.12)$$

式(8.12)中参数 X_i 是未知数,变量 a_i 是预先给定的常数。为了在一台并行计算机上尽快得到 X_i 的值,我们很容易想到使用如图 8.24(a)所示的那种树形连接结构。图中处理机排成一行,从 0 开始自左至右顺序编号。每一个后继行表明下一个时间单位要进行的工作。

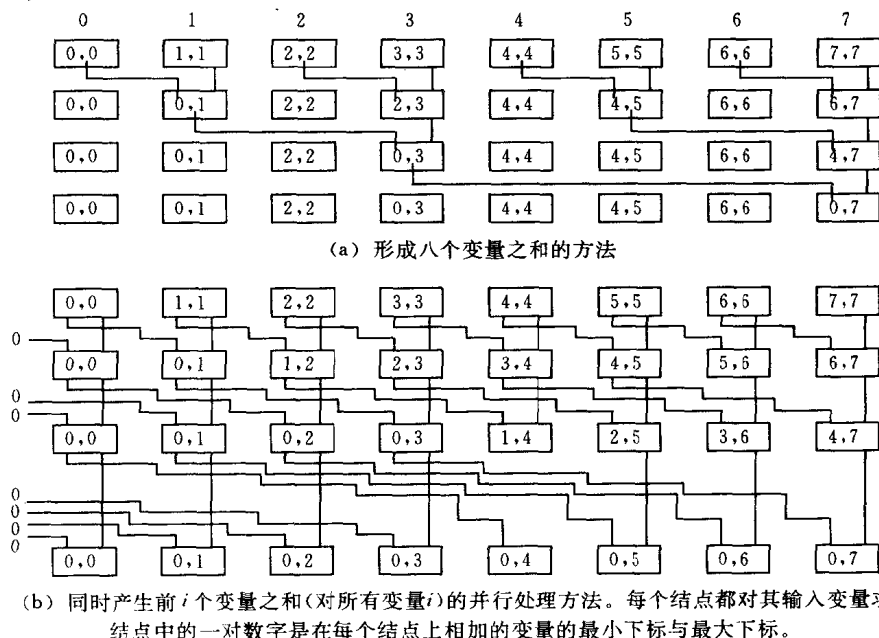


图 8.24 两种并行方法

第一次迭代,编号为奇数的处理机将其左邻送来的数值与自己的数值相加,这样编号为奇数的处理机中有了两个数相加而得的部分和。

下一次迭代,编号满足模 4 余 3 的处理机,得到来自相隔一个处理机的部分和,并把它和自己的值相加,这一次迭代产生了四个数的部分和。

第 j 次迭代时, 2^j 个数的部分和已经形成。如果 i 满足 $i = 2^j - 1$,那么在 j 次迭代结束时,我们已经计算出 X_i 的值了。由于树的深度代表计算时间,树的宽度代表迭代的长度,因为有 N 个叶子的树的深度正比于 $\log_2 N$,所以我们可以算出长度为 N 的递归计算所需时间正比于 $\log_2 N$ 。

图 8.24(a)在 $\log_2 N$ 步之后只产生一个结果,大多数处理机在大部分时间处于空闲状态,所以处理机的利用率非常低。图 8.24(b)表明了怎样用计算一个值的时间同时算出所有 X_i 的值。第一步,每台处理机获得来自其左邻处理机的数值。第二步,每台处理机获得其左边第二台处理机的数值。第 j 步,数据来自左边第 $2^j - 1$ 台处理机。如果要求送数

的处理机不存在,用0作为数据。

图中每个处理单元中都有一对数,例如“2,3”,它们表示算法在那一时刻处理单元所要相加的数的下标范围。所有的部分和都是同时形成的。

这个算法指出一种重要的互连模式,即把编号相差1、2、4、8...的处理机互相连接。由此我们知道为什么多立方体具有六维立方体的几何结构。如果我们把多立方体中的每台处理机的6位二进制标号看作是一个整数,那么每台处理机与它的标号差值是1、2、4、8、16和32的处理机相连,这里我们需要把差取绝对值,上述结论才是正确的。如果一台处理机的标号在某一位上是0,例如权是4的那一位为0,若其邻居在该位为1,这样其邻居处理机的标号看作一个整数时,它比这台处理机的标号大4。反过来,如果一台处理机在权是4的那一位为1,其邻居在该位是0,这样其邻居的标号看作一个整数时,比这个处理机的标号小4。

根据图8.24(b)描述的算法,每台处理机必须与标号比它小4的处理机相连,但多立方体中只有一半处理机能够做到这一点。不管怎样,通过间接通信,多立方体也能够形成图8.24(b)所示的树状和数,不过通信需要一定的开销。

图8.19中的树和图8.24的互连网络不只是用于求部分和这种简单的线性递归,还能用来处理其他问题。下面考虑乘法递归:

$$\begin{aligned} X_i &= a_i X_{i-1}, \quad i > 0 \\ X_0 &= a_0 \end{aligned} \quad (8.13)$$

我们再次假设 X_i 是未知量,系数 a_i 是已知常数。求解这一问题是要计算部分积,而求解式(8.12)是计算部分和。实际上只要把加法改成乘法就可以利用图8.24所示的结构把所有未知量的值同时计算出来。

求解式(8.12)和(8.13)的方法如此相似的原因是加法和乘法都具有结合律。求一组数据的和与求一组数据的积时,计算的顺序无关紧要。更确切地说,例如,串行方式计算式(8.12)的 X_3 的表达式如下表示:

$$X_3 = ((a_0 + a_1) + a_2) + a_3$$

用图8.24所示的方法计算 X_3 时好像表达式用加括号的办法重新进行了组合: $X_3 = (a_0 + a_1) + (a_2 + a_3)$ 。

如果求值的运算是可结合的,那么根据结合律的定义利用括号重新组合该表达式。结合律是很有用的,它使我们立即能够解决许多经常遇到的递归问题。其中最常用的是:

- (1) MAX(a,b)求a,b中最大的数。
- (2) MIN(a,b)求a,b中最小的数。
- (3) XOR(a,b)如果a与b是布尔变量,a与b进行异或运算。
- (4) OR(a,b)如果a与b是布尔变量,a与b进行逻辑或运算。
- (5) AND(a,b)如果a与b是布尔变量,a与b进行逻辑与运算。

现在让我们看看这一思想还能有什么新发展。考虑下面的递归式子:

$$X_i = a_i X_{i-1} + b_i \quad (8.14)$$

这个式子中不包含可结合的运算,由此初看上去前面对于可结合运算所做的分析似乎没有什么用处。但是我们可以把式(8.14)改写成另一种含有可结合运算符的形式。我

们定义 X_i 是一列矩阵: $X_i = [X_i, 1]^t$ 。其中上角标 t 表示转置。为了把式(8.14) 改写成矩阵的形式,我们再定义矩阵 A_i 为:

$$A_i = \begin{bmatrix} a_i & b_i \\ 0 & 1 \end{bmatrix}$$

利用这两个矩阵定义,式(8.14)便可以写成我们熟悉的形式:

$$X_i = A_i X_{i-1} \quad (8.15)$$

上式中 A_i 的第一行代表式(8.14),而第二行则是平凡等式 $1=1$ 。式(8.15)的重要特性是矩阵乘法符合结合律,因此这个方程的递归计算可以采用与式(8.13)的计算相同的方法进行,不过解方程(8.15)时每个结点执行的运算是 2×2 的矩阵乘法。

整个矩阵相乘并不是十分必要的,而且有可能减少运算次数。上面讨论的矩阵乘法有一些操作是无意义的,在实际应用中可以去掉,这已清楚地说明了可结合操作符的重要作用。

这里讲的递归技术可以推广到更一般的情形,递归的阶可以增加,例如增加为 2,这样式(8.14)变成:

$$x_i = a_i x_{i-1} + b_i x_{i-2} \quad (8.16)$$

为了用并行方式求解这一递归方程,我们重新定义 X_i 和 A_i 为:

$$A_i = \begin{bmatrix} a_i & b_i \\ 1 & 0 \end{bmatrix}$$

$$X_i = [x_i \quad x_{i-1}]^t$$

把这些代入式(8.16),我们可得到式(8.15)。更一般地说,一个阶为 d 的线性递归是其值依赖于前 d 个递归变量值的线性递归。其求解方法除了用 $d \times d$ 的矩阵乘法替换式(8.13)的有关运算外,其余都与求解式(8.13)的方法相同。

遗憾的是当 d 很大时算法的效率很低。对于 $d=1$,计算 N 项的并行算法所需时间正比于 $\log_2 N$,而串行算法所需时间正比于 N ,所以加速比正比于 $N/\log_2 N$ 。尽管它不如线性加速比好,但已经相当不错了。

函数 $\log_2 N$ 随 N 的增加非常缓慢,它所起的作用几乎是一个常数因子。我们可能更喜欢线性关系的加速比,而不喜欢 $\log_2 N$ 关系的加速比。但是,如果当 N 较小时我们认为 $\log_2 N$ 还可以忍受的话,那么当 N 增大时我们更能够忍受,因为 $\log_2 N$ 随 N 的变化非常缓慢。

对于高阶的递归问题,如阶为 d 的递归, $d \times d$ 矩阵乘法所需要的时间正比于 d^3 ,所以递归公式前 N 项的并行计算时间正比于 $d^3 \log_2 N$,而串行计算的时间正比于 dN ,所以加速比仅正比于 $N/(d^2 \log_2 N)$ 。显然,是因子 d^2 使得效率大大降低,因此解决递归问题的并行算法只适用于 d 比较小的情况。

我们可以把目前为止所探讨的技术推广应用到如下形式的递归:

$$x_i = \frac{a_i x_{i-1} + b_i}{c_i x_{i-1} + d_i}, \quad i > 0$$

$$x_0 = a_0 \quad (8.17)$$

我们为了解决这个递归问题,把 x_i 转化成关于 x_{i-2} 的函数,并且可以看到这个新的函数式与式(8.17)具有相同的结构,但它的系数由定义 x_i 和 x_{i-1} 的 8 个系数决定。这个函

数关系式指出怎样执行一系列运算,把8个系数合并成4个系数并且生成 x_i 关于 x_{i-2} 的函数的方程。因此,我们已经把递归过程中每一个 x 都依赖于它前一步的 x 值的方程变成了依赖于前两步的 x 值的方程了。

将8个系数合并成4个系数的方程是一种结点运算,用它代替解方程(8.13)的方法中所使用的可结合的运算符。我们重复这一迭代过程就可以根据前面四步的 x 值产生一个新的 x 。接着的一次迭代将产生一个仅依赖于前面八步 x 值的方程。每次迭代产生的递归方程依赖前面 x 值的距离增加一倍。 $\log_2 N$ 次递归迭代之后,使递归方程依赖于前 N 步的 x 值。所以我们可以得到的加速比的数量级为 $N/\log_2 N$ 。

在理想情况下,我们希望用 N 个处理机所获得的加速比为 N 。对于多个递归方程式来说,如果我们并行地计算 N 个或更多个递归,每个递归都在各自不同的处理机上串行地执行,那么很容易得到这个加速比。然而,并不总是那么凑巧需要计算的 N 个递归总是相互独立的。当我们想尽可能快地解决一个递归问题时,以上的研究表明其加速比与 $N/\log_2 N$ 成正比。由于分子有因子 $\log_2 N$,所以这里讲述的并行方法只适用于要求尽快求解单个递归的问题。

现在,我们把递归的理论用于连续模型的方程。在一维空间,泊松方程形式为:

$$x_{i-1} - 2x_i + x_{i+1} = -b_i \quad (8.18)$$

除两端的网格点满足具有特殊形式的方程外,中间的网格点都满足(8.18)方程。方程右边与网格点 i 处的电荷密度成正比,左边表示一点的电势等于它的相邻点电势的平均值(如不考虑电荷),如果考虑电荷,电荷的作用应加到相邻点所产生的影响中去。

通过以上讨论可以看出公式(8.8)和它的二维形式均可通过迭代求解。这里我们想说明快速递归法怎样直接并行地解方程(8.18)。我们把 i (从1到 $N-2$)依次代入式(8.18),再加上 i 等于0和 i 等于 $N-1$ 时的边界方程,就得到一个由 N 个方程组成的方程组。这个方程组称为三对角线方程组。顾名思义方程组的矩阵表示形式中所有非零项都位于三条对角线上。方程组的矩阵表示式为: $AX = b$

其中 A 的主对角线只含值 -2 ,主对角线的上下对角线为常数1。表达式的右边是一个向量 b ,它反映了一行上各网格点处的电荷密度情况。未知电位是向量 X ,我们可以通过求解方程组得到其值。

6. 递归加倍法与循环归约法

在这一小节,我们探讨求解三对角线方程组的两种方法,即递归加倍法和循环归约法。

递归加倍法是前面所得到的结论的推广。采用递归加倍法解三对角线方程组的步骤如下:

第一步,把矩阵 A 分解成 $LU = A$ 的形式,其中 L 和 U 均为二对角线矩阵。 L 和 U 分别代表下和上。把 A 分解成 L 和 U 的乘积有许多种方法,我们选择这样一种方法: L 是下二对角线矩阵——所有非零元素都位于主对角线和紧挨主对角线的下面一条对角线上,并且主对角线元素均为1。 U 是上二对角线矩阵——所有非零元素都位于主对角线和紧挨主对角线上面的一条对角线上。

由于上述限制,分解如果存在,那么必定是唯一的。我们总共有4条对角线,其中有一条的值都为1,于是还有三条对角线需要确定:

(1) L 的下对角线, 元素用 m_i 表示 ($i = 1, \dots, N-1$)。

(2) U 的主对角线, 元素用 U_i 表示 ($i = 0, \dots, N-1$)。

(3) U 的上对角线, 元素用 f_i 表示 ($i = 0, \dots, N-2$)。

把 A 分解成 L 和 U 的乘积之后, 我们可以写出:

$$Ax = LUx = Ly = b$$

这里我们用等式 $Ux = y$ 定义 y 。

第二步和第三步涉及二对角线方程组的求解。

已知 L 和 b , 根据 $Ly = b$ 求 y 。

已知 U 和 y , 根据 $Ux = y$ 求 x 。

这两步很容易并行计算, 因为它们具有各自的形式:

$$m_i y_{i-1} + y_i = b_i$$

$$u_i x_i + f_i x_{i+1} = y_i$$

其中 y_0 和 x_{N-1} 是边界条件。每个 y_i 满足一个类似式 (8.14) 的方程, 而且每一个这样的方程可以使用与求解式 (8.14) 相同的方法进行求解。未知量 x_i 的值可用同样的方法确定。只有一点不同, 它不是向前递归, 而是从 x_{N-1} 开始向后递归到 x_0 。计算中最困难的部分是求解矩阵 L 和 U 的对角线元素, 最难计算的是矩阵 U 的主对角线上的元素。它们满足下面的递归式:

$$U_i = a_{i,i} - \frac{a_{i,i-1}a_{i-1,i}}{U_{i-1}}, \quad 1 \leq i \leq N$$

$$U_0 = a_{0,0} \quad (8.19)$$

式 (8.19) 是式 (8.17) 的特殊情况, 所以所有的 u_i 可并行地计算出来。另外两条对角线由于存在下述关系, 很容易计算出来:

$$m_i = \frac{a_{i,i}}{U_{i,i}}, \quad 0 \leq i \leq N-2$$

$$f_i = a_{i,i+1}, \quad 1 \leq i \leq N-1$$

其中 L 的下对角线 (m_i 项) 可用 A 的主对角线向量除以 U 的主对角线向量求得。 U 的上对角线恰好就等于 A 的上对角线。

上述简要的讨论证实三对角线方程组的全部求解可以在 N 台处理机上并行地完成, 得到的加速比与 $N/\log_2 N$ 成正比。由于解决这种问题所需要的处理机之间的互连方式与我们解决连续模型问题时采用的近邻互连方式不同, 它采用图 8.24 那种互连方式, 即 N 台处理机中的每一台都直接间接地和它相隔 1、2、4、... 直至 $N/2$ 的处理机相连。

如果从处理机对处理机的影响的角度来观察图 8.24 中的数据流, 那么每一步计算使一台处理机能够影响的处理机数目增加一倍。在 $\log_2 N$ 步之后, 一台处理机就可以影响整个系统的 N 台处理机。加速比表达式中分母 $\log_2 N$ 是一个导致效率降低的因素, 它代表通信所需的开销, 因为并行通信要经过 $\log_2 N$ 步, 而串行通信则不需要。

第二种求解三对角线方程组的方法叫做循环归约。它对求解对角线值只含 1 和 -2 的泊松矩阵非常有效。循环归约方法也可以推广到其他矩阵。循环归约的主要思想是对

如下形式的三个连续方程求和：

$$\begin{aligned} x_{i-2} - 2x_{i-1} + x_i &= -b_{i-1} \\ x_{i-1} - 2x_i + x_{i+1} &= -b_i \\ x_i - 2x_{i+1} + x_{i+2} &= -b_{i+1} \end{aligned} \quad (8.20)$$

我们把式(8.20)的中间这个方程的 2 倍与其他两个方程相加,可得到:

$$x_{i-2} - 2x_i + x_{i+2} = -(b_{i-1} + 2b_i + b_{i+1}) \quad (8.21)$$

如果我们有一组方程,方程个数比 2 的幂少 1,采用上面这种方法可以得到一个新的方程组,这组方程在形式上类似于原方程组,只不过方程个数大约只有原来的一半。

我们可以对式(8.21)中的变量重新编号以便使下标连续。这样,就把归约后的方程组又变成了式(8.20)的形式。于是重复这一过程,每次从方程组中去掉大约一半的方程,直到最后留下一个可直接求解的方程。

为求解全部方程组,我们把已知的一个解代入最后一步产生的方程,就可求得另外两个未知量的值。把这三个已知解代入倒数第二步产生的方程,可以得到七个未知量的值。

重复地进行回代过程,每一步求出的未知量的个数大约增加一倍,直到求出所有的变量值。归约过程和回代过程都可以并行地进行。循环归约的加速比与递归加倍法的加速比差不多,但由于它具有较好的稳定性,所以比递归加倍法更好些。

将循环归约法和递归加倍法对通信方式的要求作个比较是很有意义的。在计算中,要进行通信的处理机的编号在第一次迭代时相差 1,接着在以后的迭代过程中依次相差 2、4...直到 $N/2$ 。在此我们又一次看到一个变量的影响范围每一步都成倍增长,在 $\log_2 N$ 步后影响已传播到整个系统。

早先我们讨论的有关连续模型的算法采用近邻互连方式,这种互连方式每一步只能将信息传播固定有限的距离,而递归加倍法和循环归约法每一步都能将信息传播的距离加倍。从通信的观点来看,直接方式比重复方式功能更强,效率更高,但是重复方式本身的功能也很强。在实际应用中,当起主要作用的因素位于附近,而较远处的影响可忽略的情况下,重复方式的效率也是比较高的。

这部分讨论的中心思想是:

- (1) 一些看起来只能串行地计算的问题可以用并行计算法解决,并有比较好的加速比。
- (2) 一个用于求解全信息问题的好的并行算法每次迭代应使每个变量的影响范围扩大一倍。
- (3) 能支持影响范围成倍扩大的通信方式是在一系列迭代过程中处理单元能依次和与之相距 1、2、4...的处理单元通信。

8.5 连续模型的结构向何处发展

连续模型是并行处理机的一种很自然的模型。接近邻互连方式连接的处理机可以模拟近邻相互作用。近邻连接的系统处理与这种结构很匹配的问题时非常有效。但处理与

这种结构不太匹配的问题时,它的性能就非常不好,并行执行的特点被硬件的低效使用所淹没了。设计系统结构时有以下几种选择:

1. 设计一种非常专用的近邻互连的机器,它处理连续模型的某些问题非常高速且非常有效。

2. 设计一种较通用的机器,它处理连续模型的问题仍有很高的速度。这种机器有较强的互连网络,如混洗以及其他一些机构,使得处理非连续模型问题的速度也比较高。

3. 设计一种非常通用的并行处理机,它的应用非常广泛。它也能解决连续模型的问题,但速度不如专为解决这类问题而设计的机器快。

从第一种选择转向第二种选择时,机器的用户数量增加 1~2 个数量级。从第二种选择转向第三种选择时,用户数量又增加 1~2 个数量级。用户越多,每个用户负担的软件开发费用就越少。

需求越大,对发展的促进也越大。但是,若设计者选择了一个用户数量很大的方案,设计了一个非常通用的系统结构,那么那些只需要解决连续模型问题的用户就不满意了。因为这种非常通用的机器解决连续模型的问题远不如专为解决连续模型问题所设计的机器快。此外,这些用户也会对那些能处理很通用问题的软、硬件的价值产生怀疑,因为他们分担了开发那些软硬件的费用,却没有从中得到任何特别的好处。

系统结构设计者到底应该为哪一类用户服务呢?这个问题还没有一个明确的回答。系统结构设计者应该准备研制任何可能的机器,包括最专用的机器和最通用的机器。每一种机器的系统结构都应该是性能价格比最优。

实际上,市场的需要和其他优先考虑将决定制造哪种机器。有些开发者可能选择最通用的方案,希望生产量很大。有些开发者可能选择一种高度专用的机器,其生产量较少。有些开发者可能选择一种介于以上两者之间的设计方案。

无论哪种选择,对用户来说,该系统结构的性能价格比应该比较高的。如果其市场较小,那么努力降低软硬件的开发费用非常重要,这样才能保证机器的售价在比较合理的范围之内。因此,系统结构设计者不仅要提供一种性能价格比高的设计方案,而且设计过程本身也必须非常有效。

从这一章我们可以得到一个很重要的结论:一种对某类问题似乎很理想的系统结构,实际上可能一点也不理想。一种专用的系统结构执行某种程序可能非常有效,但是在基本算法的研究取得进展时,可能产生一种新的,高效的技术,但却一点也不适合于原先那种专用的系统结构。遇到这种情况,专用的机器无法和那些不太专用,但能运行更有效的算法的机器竞争。各种突破随时都会发生,例如在 1964 年出现了快速傅里叶变换公式。系统结构越专用化,那么当新的突破出现时,竞争的方法就越容易受影响。专用机器的系统结构设计者必须估计到一个新的突破到来时的风险。最近几年算法的改进改变了数据的流动方式,改变了能模拟连续模型的网络结构。假设有一台为专门执行 20 年前算法而制造的机器,用它来执行当前新的算法就会显得非常差了。

总之,我们不能绝对地说哪种多处理机方案是处理连续模型问题最好的方案。采用哪种方案与处理系统的专用化程度有关。处理连续模型问题的处理机毫无疑问在某种程度

上是专用的,因此它采用了能使这类问题的程序加速执行的互连方式。究竟哪一种方案会占统治地位,这主要取决于今后几年设备技术的发展情况。例如近邻连接结构将依靠VLSI的发展,而全混洗结构将依靠互连技术的发展。

习 题 八

- 8.1 解释下列术语
并行处理机;SIMD 计算机;阵列处理机
- 8.2 评述 SIMD 计算机和向量处理机处理向量的优缺点。
- 8.3 在 Illiac N 上实现两个 8×8 矩阵乘 $C = A * B$ 。要求 J, K 循环并行完成,数据在存储器中不准重复存放,考虑到各次数据传送时间,请设计能使 64 个处理单元全并行工作的算法。
- 8.4 试设计一在 Illiac N 计算机(含 64 个 PE)上完成对 8×8 三角矩阵 $A = (a_{ij})$ 求逆的 SIMD 算法,说明实现此算法所需的存储器分配和 CU 指令。主要的设计要求是使所用的指令步数和数据存储字最少。假设给定矩阵 A 是非奇异矩阵。可对 PE 采用屏蔽,使其处于活动或不活动状态。还要求每执行一步指令列出有关 PE 寄存器和 PEM 的内容。
- 8.5 给定一幅 $n \times m$ 图象,其每个图素(象元素)的灰度级在 0 到 $b-1$ 范围内。令 $A[i, j]$ 为图素 (i, j) 上的灰度级。则在 SISD 计算机上构造一个直方图的算法如下:

```

For i=0 to b-1 do
    Histogram(i) ← 0;
For i=1 to n do
    For j=1 to m do
        Histogram(A[i, j]) ← Histogram(A[i, j]) + 1;

```

现在我们要用含 p 个 PE 的 SIMD 机来构造这个直方图,假定 $n, m \gg p, n, m$ 和 p ($p = 2^y$) 都是 2 的幂, $n/p = k$ 是一整数。每个 PEM 存放 k 行图象数据,即 PEM₀ 存放行 1 和行 k , 等等。存储格式如下图所示,图中每个 PEM 内从 a 至 $a+b-1$ 存储单元存放的是局部直方图。构造直方图的方法是先在每个 PE 中形成局部直方图,再将这些局部直方图连起来构成全局直方图。

每个 PE _{i} 能用一步与它的近邻 PE _{$i+1$} 和 PE _{$i-1$} 进行通信。请利用表 8.2 所列的向量指令和标量指令,表中每条指令下面所标的数字即为该指令的执行时间。同时,我们还假定控制器中有五个全局变址寄存器。

- (1) 试用这些指令编写一个程序,在此 SIMD 机上构造一幅直方图,结果应存入 PEM₀ 的存储单元 $a, a+1, \dots, a+b-1$ 中。
- (2) 试计算该程序所需总的周期数,以及该程序对 SISD 机(由 CU 和一个 PE 组成)上常规程序的加速比。假定每条标量指令的执行时间与它对应的向量指令的时间一样。注意,在 SISD 计算机中是没有通信开销的。

表 8.2 习题 8.5 所需的向量-标量指令系统实例

VLOAD r (3 个周期)	变址 $(r) \leftarrow (D_i) + (I_i)$ $r \in \{A_i, B_i, C_i, R_i\}$	VADDI $r, \#$ (3 个周期)	立即数相加 $(r) \leftarrow (r) + \# r \in u_i$
VMOV r_1, r_2 (1 个周期)	寄存器传送 $(r_1) \leftarrow (r_2)$ $r_1, r_2 \in U_i$ $U_i = \{A_i, B_i, C_i, R_i, I_i, R_i\}$	LCYCLE r (s 个周期)	左循环移数 $(R_{i-1}) \leftarrow (R_i)$ $s = 2^d, d = (r)$ $r \in \{A_i, B_i, C_i\}$
VMVI $r, \#$ (2 个周期)	立即操作数 $(r) \leftarrow \#$ $r \in U_i$	MVI INX, $\#$ (1 个周期)	立即操作数 $(\text{INX}) \leftarrow \# \text{inx} \in z$ $z = \{\text{INX}_1, \text{INX}_2, \dots, \text{INX}_5\}$
VSTORE r (3 个周期)	变址 $((D_i) + (I_i)) \leftarrow (r)$ $r \in \{A_i, B_i, C_i, R_i\}$	ICR INX, 1 (1 个周期)	变址寄存器增值 $(\text{INX}) \leftarrow (\text{inx}) + 1$ $\text{INX} \in z$
VADD r_1, r_2 (2 个周期)	寄存器相加 $(r_1) \leftarrow (r_1) + (r_2)$ $r_1, r_2 \in U_i$	JLTINX _i , INX _i , LOOP (2 个周期)	条件分支 若 $(\text{INX}_i) < (\text{OMX}_i)$ 则转 LOOP $\text{INX}_i, \text{INX}_j \in Z$

地址	PEM ₀	PEM ₁	PEM _{p-1}
0	A[1,1]	A[k+1,1]	A[n-k+1,1]
1	A[1,2]	A[k+1,2]	A[n-k+1,2]
	⋮	⋮	⋮
km-1	A[k,m]	A[2k,m]	A[n,m]
	⋮	⋮	⋮
a	h ₀ (0)	h ₁ (0)	h _{p-1} (0)
a+1	h ₁ (0)	h ₁ (1)	h _{p-1} (1)
	⋮		
a+b-1	h ₀ (b-1)	h ₁ (b-1)	h _{p-1} (b-1)
	⋮	⋮	⋮

图 8.25 习题 8.5 中构造直方图的数据存储

- 8.6 试在含一个 PE 的 SISD 机和在含 m 个 PE 且连接成一线性环的 SIMD 机上计算下列求内积的表达式。

$$s = \sum_{i=1}^n A_i \cdot B_i$$

假定完成每次 ADD 操作需 2 个单元时间,完成每次 MULTIPLY 操作需 4 个单元时间,沿双向环在相邻 PE 间移数需 1 个单元时间。

- (1) SISD 计算机上计算 s 的时间是多少?
 - (2) SIMD 计算机上计算 s 的时间是多少?
 - (3) 用 SIMD 机计算 s 相对于用 SISD 机计算的加速比是多少?
- 8.7 今有 K 对向量,其中第 i 对由行向量 R_i 和列向量 C_i 组成,每个的维数为 $N = 2^n$ 。可按下式计算第 i 对向量的内积:

$$IP[i] = \sum_{j=1}^N R_i[j] \cdot C_j[j]$$

下面是完成 $IP[i] (i = 1, 2, \dots, K)$ 的算法

```

For i ← 1 to K do
begin
  IP[i] ← 0;
  For j ← 1 to N do
    IP[i] ← IP[i] + R_i[j] * C_j[j];
  end
end

```

- (1) 忽略初始化、下标修正和测试等所需的时间, 试计算在单处理机上实现上述算法总共需多少时间, 并表达成 K 和 N 的函数, 假定完成乘法与加法需用相同的单位时间。
 - (2) 为加速上述计算, 可采用 SIMD 机来发掘计算中的并行性, 试求出下列两种不同情况上的计算时间。
 - (i) 用 $P = N$ 个处理单元 PE 逐对地计算每对 R_i, C_i 的 $IP[i]$ 。
 - (ii) 将一对向量分配给每个 PE, 由此 PE 来计算其内积。在这种情况下 PE 数 $P = K$ 。
- 8.8 我们已经知道, 假如错开距离为 $S = 2^n$, 则含 $P = 2^{2n}$ 个 PE 的 SIMD 机能从 $M = 2^{2n} + 1$ 个并行存储器模块无冲突地对一个矩阵按行、列、对角线和反对角线进行访问。试证明在同样条件下系统能用一个存储周期对任何一个 $2^n \times 2^n$ 方块进行访问。
- 8.9 下面是一个 8×8 矩阵 A 在有 $M = 5$ 个存储器模块和 $P = 4$ 台处理机的阵列处理机中的错开存储分配方案。
- (1) 试列出在一个存储周期内可以进行访问的全部模式
 - (2) 写出元素存放的规则, 即元素 $A(i, j)$ 与模块内地址 A_i 及模块号的一般关系。

模块内 地址 Ad	模块				
	M_0	M_1	M_2	M_3	M_4
0	00	10	20	30	×
1	50	60	70	×	40
2	21	31	×	01	11
3	71	×	41	51	61
4	×	02	12	22	32
5	42	52	62	72	×
6	13	23	33	×	03
7	63	73	×	43	53
8	34	×	04	14	24
9	×	44	54	64	74
10	05	15	25	35	×
11	55	65	75	×	45
12	26	36	×	06	16
13	76	×	46	56	66
14	×	07	17	27	37
15	47	57	67	77	×

注: 表中数字对应于下标 ij 。

- 8.10 在 16 台 PE 的并行处理机上,要对存放在 M 分体并行存储器中的 16×16 二维数组实现行、列、主对角线、次对角线上各元素均无冲突访问,要求 M 至少为多少?此时数组在存储器中应如何存放?写出其一般规则。同时证明这样存放同时也可以无冲突地访问该二维数组中任意 4×4 子阵列的各元素。
- 8.11 一台向量计算机只能以下述两种方式中的一种运行:一种是向量方式,执行速度 R_v 为 10 Mflops;另一种是标量方式,执行速度 R_s 为 1 Mflops。设 a 是该计算机的典型程序代码中可向量化部分的百分比。
- (1) 推导出该计算机平均执行速度 R_a 的公式。
 - (2) 画出以 a 为横坐标, R_a 为纵坐标的曲线, a 的范围为 $(0, 1)$ 。
 - (3) 要使 R_a 达到 7.5 Mflops,问向量化百分比 a 应多大?
 - (4) 假设 $R_s = 1$ Mflops, $a = 0.7$, 要使 R_a 达到 2 Mflops,问 R_v 应为多大?
- 8.12 设计一种采用加、乘和数据寻径操作的算法,分别在下面两种计算机系统上用最短的时间来计算表达式 $s = A_1 \times B_1 + A_2 \times B_2 + \cdots + A_{32} \times B_{32}$ 。假设加法和乘法分别需要两个和四个单位时间,从存储器取指令、取数据、译码的时间忽略不计,所有的指令和数据已装入有关的 PE。试确定下列每种情况的最小计算时间。
- (1) 一台串行计算机,处理机中有一个加法器和乘法器,同一时刻只有其中一个可以使用。这种单处理机系统不需要数据寻径操作。
 - (2) 一台有 8 个 PE (PE_0, PE_1, \dots, PE_7) 的 SIMD 计算机,8 个 PE 连成双向环结构。每个 PE 用一个单位时间可以把数据直接送给它的相邻 PE。操作数 A_i 和 B_j 最初存放在 $PE_{i(\text{mod } 8)}$ 中,其中 $i = 1, 2, \dots, 32$ 。每个 PE 可在不同时刻执行加法或乘法。
- 8.13 $A = (a_{ij}), B = (b_{ij})$ 为 64×64 的矩阵。有一台由 64 个带本地存储器的 PE 组成的 SIMD 机器,64 个互连成 8×8 的两维双向链路的环网结构。试设计一种矩阵乘法算法,使之在这种机器上执行的时间最短。
- (1) 画出矩阵元素 a_{ij} 和 b_{ij} 在 PE 存储器上的初始化分配情况。
 - (2) 确定完成矩阵乘法所需要的 SIMD 指令。假设每个 PE 在每个周期可以执行一次乘法,或一次加法,或一次移数(把数据移给它四个相邻之一)操作。在把数据传给相邻 PE 之前,应该首先对本地数据进行乘法和加法操作。SIMD 移数操作可以在循环连接的环网上或向东、或向西、或向南、或向北进行。
 - (3) 估算矩阵乘法总共需要多少个 SIMD 指令周期,这包括所有的运算和数据寻径操作时间在内。最后乘积 $C = A \times B = (C_{ij})$ 在各个 PE 的存储器中没有拷贝。
 - (4) 假设开始时允许数据拷贝,即同一数据元素装入多个 PE 的存储器。试设计一种新的算法,以进一步减少 SIMD 执行时间。这时必须将通过数据广播指令或数据寻径(移数)指令进行原始数据拷贝的时间考虑在内,而且每个结果元素 c_{ij} 只放在每个 PE 的存储器中。
- 8.14 BSP 是一台由 16 个 PE 和 17 个共享存储器模块组成的 SIMD 计算机。试证明如果跳距不为 17 的倍数,那么对任意长度向量的访问都不会出现存储器冲突。

第九章 多处理机

前几章讨论了如何加快单指令流执行速度的方法。尽管只有一个程序在执行,但前面讨论的各种技术已涉及了开发指令流内部或一条指令内部的并发性问题。本章讨论多处理机——由若干台独立的处理机组成的系统。由于器件本身限制了任何单处理机的速度最高不能超过某个上界值,欲超越这个值就需要研究多处理机系统。

本章的中心议题是多处理机的结构和性能。将介绍把多台处理机组成高并行度系统的有关技术,并深入分析这类系统的瓶颈和改进性能的方法。下一章将讨论多处理机系统所要采取的软件策略问题。

9.1 多处理机结构

9.1.1 两种多处理机结构

前面讨论了两类并行度的机器,即流水线机器和连续计算的机器。流水线机器通过几级流水的同时操作来获得高性能。连续计算的机器由多台处理机组成,每台处理机执行相同的程序。上述两种机器都是执行单个程序,对向量或数组进行运算。Flynn 将这类并行度称为单指令流多数据流(SIMD)并行度。这种系统结构能高效地执行适合于 SIMD 的程序,所以这种机器对某些应用问题非常有效。

为了发挥这类机器的高性能,需要重新改写传统的算法,希望改写后的算法能把指令播送给所有处理机来同时处理大量数据。虽然面向这种机器的程序设计的原理非常困难,但在理想情况下,只要将串行算法中每一内循环用一条能实现全循环的播送指令替代就转换成 SIMD 算法了。有一类很重要的应用问题非常适合于上述这种模型,这就促使人们去设计和构造这类机器。

但是,有些大型题目在这种 SIMD 结构机器上运行不那么有效。原因是这类问题没有对结构化数据进行重复运算的操作,它所要求的操作通常是非结构化并且不可预测。寻址方式通常与数据相关,所以系统结构很难通过预测未来的访问把数据提前装入。

要想解决这类问题并保持高性能,系统结构设计者只能在多处理机结构中寻找解决的方法。多处理机结构由若干台独立的计算机组成,每台计算机能够独立执行自己的程序。Flynn 称这种结构为多指令流多数据流(MIMD)结构。多处理机系统中的处理机之间按某种形式互连,从而实现程序之间的数据交换和同步。

多指令流多数据流结构如图 9.1 所示。图中每台处理机都有寄存器、运算器、逻辑部件、访问存储器和 I/O 设备的通道。图 9.1(a)中存储器和 I/O 设备是独立的子系统,为所有处理机共享。图 9.1(b)中每台处理机有自己的存储器和 I/O 设备。图 9.1 的系统有多台处理机,每台处理机能够执行独立的程序,所以属于 MIMD 类机器。在图 9.1 所示的系统中,处理机通过互连网络交换数据和实现同步。图 9.1(a)中的存储器共享是实现信息

交换和同步最简单的方法,任何两台处理机可以通过共享存储器的单元实现通信。图 9.1 (b)这种结构通过点对点的信息交换实现通信。显然,多处理机系统可以采用共享存储器方案或本地存储器方案,或这两种方案的结合。

图 9.1 可以看出多处理机在结构原理上区别于并行处理机的主要特点。第一,它有多多个控制器,至少有多多个指令部件,用以对各个 PE 实现单独的控制,而又相互协调配合。第二,多处理机的外围设备要能够被多个 PE 分别调用,因而要通过互连网络转接,而不像并行处理机的外围设备那样统一访问主存储器进行程序和数组的有规则的传送。第三,并行处理机由于主要完成数组向量运算,它的 PE 和 MM 之间的数据交往是比较有规则的,存储器访问的地址变换功能不要求太高,因而互连网络的作用主要放在数据对准上,可以做得比较简单。但是,多处理机由于互连网络必须满足各个 PE 随机地访问主存储器的要求,所以,连接模式、频带和路径选择等问题都要复杂得多。存储映射部件对每一个 PE 也是必需的。这是由于在多处理机系统中,存储器的数据和存储空间都要被多个处理机共享,不能允许每一个 PE 在运行自己的程序段时直接产生物理地址,从而要利用存储映射部件来满足存储器动态分配和处理机共享数据块的需要。

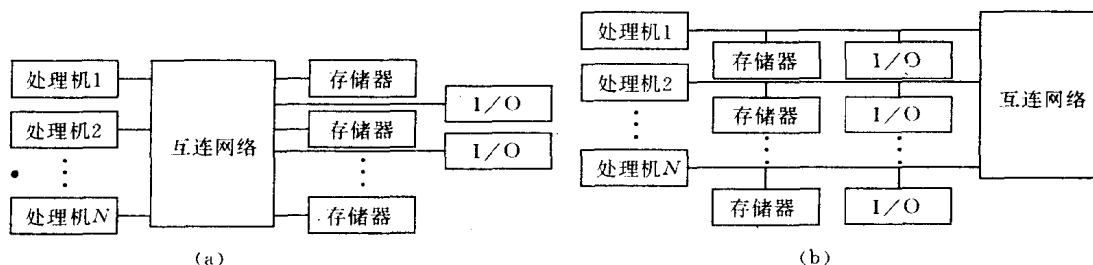


图 9.1 两种多处理机结构

使用高速多处理机的主要目的是利用多台处理机并发地执行一个作业,使得执行速度比单处理机快。在某些应用场合,使用多处理机的主要目的是高可靠性而不是高性能。如果某台处理机出现故障,那么它的程序可以由系统中其它处理机来执行。这种系统的设计原理和为获得高性能的多处理机设计原理有很大的不同,本书不准备讨论为求高可靠性的多处理机系统。

9.1.2 多处理机系统的特点

多处理机属于多指令流多数据流(MIMD)计算机,它和上一章叙述的、属于单指令流多数据流(SIMD)计算机的并行处理机相比,有很大的差别。它们的差别归根到底来源于并行性级别的不同:多处理机要实现任务一级的并行,不能再像 SIMD 计算机那样只能对多数据流执行同一指令操作。因此,在结构上,它的多个处理机要用多个指令部件分别控制,并且要有复杂的互连网络实现机间通信;在算法上,不限于数组向量处理,要挖掘和实现更多通用算法中隐含的并行性;在系统管理上,要更多依靠软件手段有效地解决资源管理,特别是处理机管理以及进程调度等问题。下面概括说明多处理机系统的特点。

1. 结构灵活性

并行处理机是从解决专用题目的需要而发展起来的。其结构主要是针对数组向量处

理算法而设计的。结构特点是：处理单元很多，但只需设置有限和固定的机间互连通路，即可满足一批并行性很高的算法的需要。虽然发展了像 BSP 这样通用性较强的系统，但从并行处理机来说，仍是以解决大型数组计算问题为主。用库克的分类标准，它仍是属于数组单执行(SEA)一类。而多处理机则有所不同，它应有较强的通用性。例如：它可以同时对多个数组进行不同的处理。这称为数组多执行(MEA)，亦可称为多 SIMD(MSIMD)；它可以同时对多个标量数据进行不同的处理，这称为标量多执行(MES)；等等。这就要求多处理机能适应更为多样的算法，具备更为灵活多变的系统结构以实现各种复杂的机间互连模式，同时还要解决共享资源的冲突问题。目前，多处理机中处理单元的数目还不可能做得很多。

2. 程序并行性

并行处理机实现操作一级的并行，其并行性存在于指令内部，一条指令可以同时对整个数组进行处理，再加上系统具有的专用性特点，就使程序并行性的识别较易实现。这一点主要在指令类型及硬件结构上考虑，可由程序员在编制程序中加以掌握，或由向量化编译程序协助。但在多处理机中，因为不限于解决数组向量处理问题，并行性存在于指令外部，即表现在多个任务之间，再加上系统通用性的要求，就使程序并行性的识别难度较大。因此，它必须利用多种途径，如算法、程序语言、编译、操作系统、以至指令、硬件等，尽量挖掘各种潜在的并行性，而且主要的任务不能放在程序员肩上。

3. 并行任务派生

并行处理机依靠单指令流对多数据流实现并行操作，这种并行操作是通过各条单独的指令加以反映和控制的，这样由指令本身就可以启动多个处理部件并行工作。但多处理机是处于多指令流操作方式，一个程序当中就存在多个并发的程序段，需要专门的指令来表示它们的并发关系以控制它们的并发执行，以便一个任务开始被执行时就能派生出可与它并行执行的另一些任务。这个过程称为并行任务派生。派生的并行任务数目是随程序和程序流程的不同而变化着的，并不需要多处理机系统用固定数目的处理机加屏蔽的方法来满足其执行的需要。多处理机执行这些并行任务，不必浪费多余的处理机，需要多少就分配多少，如果不够，那些暂时不能分配到空闲处理机的任务就进入排队器，处于等待状态。这样就使多处理机有可能达到较高的效率。这是它较之并行处理机具有的潜在优点。

4. 进程同步

并行处理机实现操作级的并行，所有处于活动状态的处理单元同时执行共同的指令操作，受同一个控制器控制，工作自然是同步的。但多处理机所实现的是指令、任务、程序级的并行。一般说，在同一时刻，不同的处理机执行着不同的指令。由于执行时间互不相等，故它们的工作进度不会也不必保持相同。并行任务被派生以后，要根据分配到空闲处理机的先后次序陆续投入运行，因而开始执行的时刻也不可能一致。还要由进程之间的数据相关和控制依赖决定它们执行时的正确顺序。在并行处理机看来，由于执行单指令流，程序的性质基本上是串行的，所以，虽也有指令重叠，但要遵守进程之间的正确顺序是比较容易的。而在多处理机中，情况就要复杂得多，特别是要区分进程之间的多种不同的依赖关系。如果并发进程之间有数据交往或控制依赖，那么，执行过程中有的进程就要中途

停下来进入等待状态,直到它所依赖的执行条件满足为止。这就要求多处理机采取特殊的同步措施,才能使并发进程之间保持程序所要求的正确顺序。

5. 资源分配和进程调度

并行处理机主要执行数组向量运算,处理单元数目是固定的,且受同一控制器的控制,这是程序员编写程序的基本出发点。程序员只能利用屏蔽手段来设置部分处理单元为不活动状态,以改变实际参加操作的处理单元数目。但多处理机执行并发任务,需用处理机的数目没有固定要求,各个处理机进入或退出任务的时刻互不相同,所需共享资源的品种、数量又随时变化。由于上述情况十分复杂,于是,就提出了一个资源分配和进程调度问题。这个问题解决的好坏对整个的效率有很大的直接影响。

9.2 多处理机性能模型

当多处理机系统以峰值速度运行时,所有处理机都在做着有用的工作,没有一台处理机处于空闲状态。这种情况下,多处理机系统中的 N 台处理机对系统的性能都有贡献,系统的处理速度随着 N 的增加而增加。

实际上,峰值性能是很难达到的。引起性能下降的原因是:

- (1) 由于处理机之间通信而产生的延迟;
- (2) 一台处理机与其它处理机同步所需的开销;
- (3) 当没有足够多任务时,一台或多台处理机处于空闲状态;
- (4) 由于一台或多台处理机执行无用的工作;
- (5) 系统控制和操作调度所需的开销。

串行计算机的开销主要是调度和同步。下面我们研究上述因素是怎样降低多处理机性能的。

一台高性能的向量处理机是可以避免上述许多因素的,但也会损失一些性能,因为不能保证所有处理单元都处于忙状态。特别是当一个计算问题不能转化为一组向量操作时,这个问题就尤为突出。

多处理机系统的设计者和制造者要密切注意上面提到的各种导致性能下降的因素。这些因素可能使系统的性能下降得很严重。例如,由于上述各种各样的因素使得一个系统的实际效率竟只有峰值效率的 10%,那么一个由 10 台处理机组成的多处理机系统仅仅做了相当于一台处理机所做的工作。

幸运的是,对一个处理机数目较少的系统,如果精心设计,那么性能下降的值可以较小,但是随着处理机数目的增加,性能下降会逐渐增加。存在着这么一个点,如果再增加处理机数目,系统的计算时间反而增加而不是缩短。

目前市场上可买到的多处理机系统一般是由少量的几台处理机组成,例如 4、8 或 16 台处理机,其根本原因是随着处理机数目的增加,多处理机系统的工作效率会降低。因此,目前高速的系统一般由高速的器件构成,而处理机的数目较少。

例如 Cray XMP 系统由 4 台 Cray 1 组成。IBM 309X 系列的机器由 1 台至 6 台处理机构成。这两个系统都是通过采用非常高速的器件,采用 Cache 存储器和流水线技术来获

得非常高的性能的。

这些机器的用户一般需要比单机的功能更强的系统才能解决他们的问题。由于单机的性能受到系统结构和器件技术的限制,所以在单机上再要提高性能不太容易了。一个有效的方法是把两台或四台相同的处理机连在一起获得所要求的性能。

有些计算机系统结构设计者注意到曾在第一章讨论过的成本问题。第一章讨论表明了采用高速器件的成本要比采用低速器件的成本高。

此外,目前的情况是采用高速器件会使整个系统的成本增加很多,而能得到的性能没有这么多。因此,高速机器的单位计算能力的成本要比低速机器高。虽然这一趋势还与系统结构所采用的技术有关,并且会随时间而变化。因为低速器件技术有成本低廉的优势,所以就有可能把成千上百台低速低成本的处理机连结成一个性能价格比很好的系统。

采用低成本技术会带来成本上的好处,但是系统的效率随着处理机数目的增加而降低。如果系统由于处理机数目太多而效率降低得非常厉害,那么用上百台低速处理机构成一个系统还不如用少量几台高速处理机构成一个系统。

此外,几百台处理机构成的系统的程序设计要比单处理机或少量几台处理机构成的系统的程序设计复杂。所以,虽然用几百台低速处理机构成的系统在经济上很有吸引力,但是如果效率很低,那么这种系统结构的优点也就不存在了。

由此可见,多处理机系统的并行性没有什么特别的魅力,只有当并行性带来较高的性能时,它才产生效益。如果并行性不能被有效地发掘出来,那么它只会增加系统的成本和复杂性。出现这种情况时,只有降低并行性直到能获得效益为止。

这一节我们将分析多处理机系统的性能和由于并行性所带来的额外开销问题,重点是介绍多处理机系统性能的几种模型。

从这一节我们可以看到,很大程度上性能依赖于 R/C 比值,其中 R 代表程序的执行时间, C 代表用于通信的开销,该比值表示每一单位计算的开销。如果 R/C 比值很小,那么开发并行性不会得到什么好处。如果 R/C 比值很大,那么开发并行性会得到好处。如果把一个计算作业分解成较大的作业块,那么就能得到较大的 R/C 比值,但是所得到的并行性比最大可能的并行性要小得多。

R/C 比值是衡量任务粒度(task granularity)大小的尺度。在粗粒度(coarsegrain)并行情况下, R/C 比值比较大,每个单位计算只需要少量的通信。在细粒度(finegrain)并行情况下, R/C 比值比较小,每个单位计算有很大的通信量和其它的开销。通常,细粒度并行性需要许多台处理机,而粗粒度并行性只需较少台数的处理机。细粒度并行性的基本原理是把一个程序尽可能地分解成能并行执行的小任务。在极端情况下,一个小任务只完成一个操作。通常,一个小任务包含几条指令。

程序员为了获得最高的性能,总是想方设法把一个问题分成尽可能小的粒度,以便得到最大的并行性。但是如果最大的并行性同时带来了最大的开销,那么很显然最大并行性并不一定是最佳的解决问题方法。

下面将提出一系列不同的性能模型,其主要目的在于揭示 R/C 比值的大小对性能好坏的影响是普遍的。下面的讨论将说明细粒度分配所产生的性能比粗粒度分配差的原因。虽然细粒度分配有较高的并行性,但不一定使系统有较高解决问题的速度。之所以提出

一系列不同的性能模型的原因,在于到目前为止还没有一种模型能完全反映所有多处理机结构和多处理机算法的特征,所以我们提出一系列不同的模型,以便能反映不同程序和不同多处理机系统结构的特征。不管哪一种模型, R/C 的作用是相同的,即较低的 R/C 比值由于较大的额外开销而导致较低的性能;较高的 R/C 比值其并行性开发较低。欲获得最佳性能,有必要对并行性和额外开销进行综合考虑。模型之间唯一差别在于并行性和额外开销这两个参数达到平衡的程度不同。

下面将介绍性能模型。为了简化模型,我们忽略了系统中的同步和竞争。由于这些忽略会使预测的性能比系统实际性能好一些。在大多数情况下,可以采用增大任务粒度来弥补由于以上忽略而造成的影响。

9.2.1 基本模型

假设有一个包含 M 个任务的应用程序,我们希望在由 N 台处理机组成的系统上以最快的速度执行这个程序。为了简单起见,我们先考虑一个仅有两台处理机的系统,然后再逐步增加处理机数目。为了模拟性能,我们需要用公式表示出执行时间和额外开销。

我们先承认下面两个假设是成立的,以获得初步结论,然后放宽假设,观察性能如何变化。

(1) 每个任务的执行时间为 R 个单位。

(2) 当两个任务不在同一台处理机上时,其通信所需的额外开销为 C 个单位时间。当两个任务在同一台处理机上时,通信所需的额外开销为 0。

一个应用程序在两台处理机系统上运行有多种分配方法。我们可以把全部任务都分配给一台处理机而另一台空闲,这种分配方法的通信开销最小,但没有利用并行性。我们也可以按各种不同比例将任务分配给两台处理机,那么总处理时间是执行时间和额外开销时间之和。我们用 C 表示用于通信的时间,其实它还包括系统所有其它额外开销。

在某些情况下,系统的额外开销的操作可以与计算过程重叠进行,例如处理机在执行指令的同时能通过 I/O 接口进行通信。当然并不是所有的额外开销都可以被屏蔽掉,例如处理机在访问共享数据或通信通路时可能会发生竞争,处理机在等待同步信号期间处于空闲状态等。因此,我们假设一部分额外开销的操作会增加总处理时间。在这种情况下,可以用下列等式表示一个程序的总处理时间:

$$\text{总处理时间} = R\max(M - K, K) + C(M - K)K \quad (9.1)$$

等式(9.1)表示总处理时间是两个时间的和,一个是执行时间,另一个是用于通信和其它额外开销的时间。对两台处理机系统来说,执行时间取两台处理机执行时间较大的一个。因此当 K 个任务分配给一台处理机,剩下的 $(M-K)$ 个任务分配给另一台处理机时,执行时间取 $R(M-K)$ 和 RK 中较大的一个。我们称 K 为任务分配参数。第二项表示额外开销时间与通信次数成正比例关系。通信次数与任务分配方法有关。从(9.1)式可知,第一项是 K 的线性函数,第二项是 K 的二次函数。

从等式(9.1)可知总处理时间是 K 的函数,那么其最小值是多少呢? 也就是说,任务如何分配才能获得最小的总处理时间? 我们可以采用如图 9.2 所示的图解法来求最小处

理时间。

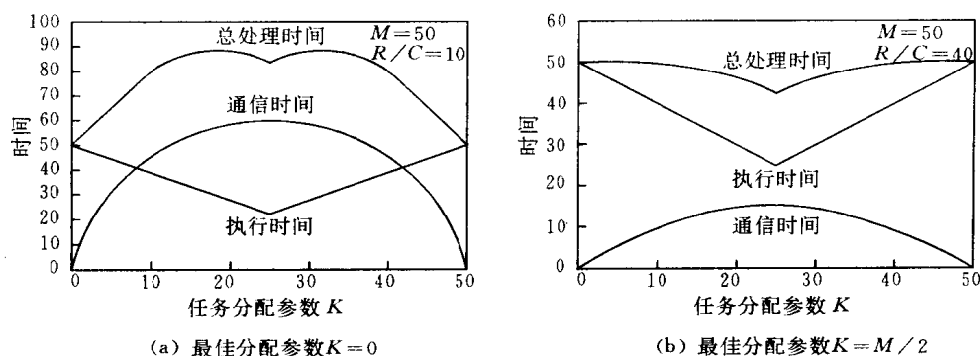


图 9.2 两种不同 R/C 比值的并行执行时间

对于该模型的结论是:当 $R/C < M/2$ 时,把所有任务分配给同一台处理机能使总处理时间最小;当 $R/C > M/2$ 时,把任务平均地分配给两台处理机能使总处理时间最小。也就是说使任务分配参数 $K=0$ 或 $K=M/2$ 。当 M 为奇数时,应使 K 尽可能接近 $M/2$ 。图 9.2 的横坐标是任务分配参数 K ,纵坐标是总处理时间。图 9.2(a)和(b)分别表示 R/C 小于 $M/2$ 和 R/C 大于 $M/2$ 时任务分配参数 K 与总处理时间的关系。等式(9.1)的第一项执行时间是分段线性的,在图 9.2 中这一项看起来好像字母 V,它对称于 $K=M/2$ 。图 9.2(a)中,由于二次曲线 $K(M-K)$ 的开口朝下,叠加一个线性项后开口仍朝下,所以最小值一定在区间 $0 \leq K \leq M/2$ 的端点处,即 $K=0$ (或 $K=M$) 时为最小值。图 9.2(b)中,当分段线性项与二次函数项叠加后,在 $K=M/2$ 处为最小值。

9.2.2 N 台处理机系统的基本模型

现在我们讨论有 N 台处理机系统的基本模型。在这种情况下,我们将 K_i 个任务分配给第 i 台处理机。等式(9.1)可推广为:

$$\begin{aligned} \text{总处理时间} &= R \max(K_i) + C/2 \sum_i K_i(M - K_i) \\ &= R \max(K_i) + C/2 \left(M^2 - \sum_i K_i^2 \right) \end{aligned} \quad (9.2)$$

从等式(9.2)的第一项求出 N 台处理机中最大执行时间。第二项计算出 K_i 与 $(M-K_i)$ 任务之间两两通信的开销时间。与等式(9.1)相同,第二项是关于 K 的二次函数。

如果分析等式(9.1)的理由对等式(9.2)也成立,那么我们可以预计等式(9.2)的最小值仍在某种极端分配情况,而实际情况也的确如此。即或者将所有的任务都集中在一台处理机上,或者将任务平均分配给所有处理机。所谓平均是指如果 M 是 N 的倍数,则每台处理机分得 M/N 个任务,否则除一台处理机外其它处理机分得 M/N 个任务,那一台处理机分得剩余的任务。这种分配并不一定使 N 台处理机都分得任务。例如,有 19 个任务和 6 台处理机,分配方法可以是:4 台处理机每台分得 4 个任务,第 5 台处理机分得 3 个任务,而第 6 台处理机什么也没有分到。

为了说明平均分配能使程序的总处理时间最小,我们假定 K_1 为分配任务中的最大

值,根据等式(9.2),可以发现通过对两台任务数小于 K_1 的处理机重新分配,使额外开销降低。假设 K_2, K_3 满足 $K_1 > K_2 \geq K_3 \geq 1$,我们将第3台处理机的某项任务移到第2台处理机,等式(9.2)的第一项的值保持不变,因为这一移动并不影响最大任务数,而第二项的值减少 $C(K_2 - K_3 + 1)$ 。可见这种分配会有更好的性能。我们可以重复这一过程,直到除一台处理机外,其它处理机上的任务数都小于最大值为止。

和等式(9.1)一样,等式(9.2)也有一个决定采用平均分配还是采用集中分配的临界值,并且等式(9.2)和等式(9.1)的两个临界值完全一致,即当 $R/C > M/2$ 时采用平均分配方法,当 $R/C < M/2$ 时采用集中分配方法。任务均分给 N 台处理机和任务集中在一台处理机,其总处理时间的差可以由下式表示:

$$\text{总处理时间差} = RM/N + CM^2/2 - CM^2/2N - RM \quad (9.3)$$

式中前三项是平均分配时的总处理时间,后一项是所有任务集中于一台处理机时的总处理时间。为了简单起见,我们假设 M 为 N 的倍数。为了计算决定采用平均分配法还是采用集中分配法的临界值,我们使等式(9.3)等于0,约去 M 后,分别以 R 和 C 为系数进行合并,再约去 $(1 - 1/N)$,等式则变为:

$$CM/2 - R = 0 \quad (9.4)$$

$$\text{或} \quad R/C = M/2 \quad (9.5)$$

这就说明,如果 R/C 比临界值 $M/2$ 大,将任务平均分配给尽可能多的处理机进行处理,能获得最短处理时间。另一方面,如果 R/C 比临界值 $M/2$ 小,即使有很多台处理机可供使用,也不可能比用一台处理机处理全部任务来得快。后一种情况需要很大的额外开销。除非额外开销低于总处理时间的某个百分比,否则并行执行不可能得到什么好处。如果本模型能够反映并行算法和并行系统结构,那么控制额外开销是保证并行性成功的绝对条件。

尽管上面分析的着眼点是性能而不是成本,但 R/C 比值的大小决定了采用哪种分配方法能使并行系统有价格优势。即使 R/C 值足够大它能保证高并行性,其性能还会由于式(9.2)中的第二项而降低。并行系统的加速比是一个计算问题在一台处理机上的运行时间与在并行系统上的运行时间(即式(9.2)的总处理时间)的比值,可近似如下:

$$\begin{aligned} \text{加速比} &= \frac{RM}{\left(\frac{RM}{N} + \frac{CM^2}{2} - \frac{CM}{2N}\right)} = \frac{R}{\frac{R}{N} + \frac{CM(1 - 1/N)}{2}} \\ &= \frac{\frac{RN}{C}}{\frac{R}{C} + \frac{M(N - 1)}{2}} \end{aligned} \quad (9.6)$$

如果分母中的第一项远远大于第二项,即 M, N 较小, R/C 较大,那么加速比与 N 成正比例。如果处理机台数 N 很大,则分母主要由第二项决定,那么加速比与 R/CM 成正比例,而不依赖于处理机的台数了。因此,随着 N 增大,加速比趋近于一个常数。这时如果再增加处理机,所提高的性能小得可以忽略,只会增加系统的成本。即使随着处理机增加,系统的性能有所改善,与所增加的成本相比也是不值得的。所以处理机的台数不应超过由成本与 R/C 比值函数所决定的极大值。

该模型说明了任务粒度与额外开销如何影响多处理机系统的性能。同时也指出了降低额外开销与合理选择粒度的重要性。然而它仅是一个模型,无论如何不能包括所有的实际应用问题。

下面还要介绍另外几种性能模型。我们会发现无论哪种模型, R/C 的大小总起着关键作用。从前面的讨论我们已经知道,如果存在最优的解决问题方法,那么极端分配方法是最好的方法,即 R/C 比值决定使用所有可以使用的处理机或只使用一台处理机。而对某些模型,这种极端分配方法并不一定是最好的方法。这类模型的最佳方法可能是将作业分配给部分处理机而不是全部处理机,因为使用过多的处理机只会降低性能和增加额外开销。在一般情况下,作业并不一定要平均分配才能获得最佳性能。

9.2.3 随机模型

下面我们考虑各个任务运行时间不相等时会发生什么情况。当所有处理机的运行时间相等时,等式(9.2)中的第一项为最小值。所以,我们总设法将任务分配给所有处理机,使所有处理机有相等的运行时间。如果不能相等,处理机中最长运行时间也应尽可能地短。

尽可能不平均地分配任务会使等式(9.2)中的第二项为最小值。因此,在使各台处理机运行时间尽可能相等的前提下,使任务分配尽可能不均衡。也就是说,要寻找这样一种分配方案,既使每台处理机分配到的任务尽可能多或尽可能少,又要使每台处理机的总工作负荷超过某个规定值。

在此模型中,最均衡地分配工作负荷不一定是最佳分配。工作负荷轻微的不平衡有可能使系统额外开销大大减少。也就是说,等式(9.2)中第一项线性函数的小增量可以导致第二项二次函数的急剧下降。

假设 R 为运行时间, C 为通信时间,且 R 和 C 不是固定常数,它们是独立的随机变量。为了解释这个模型,借助于中值定理我们进一步假设:

$$E\left[\max\left\{\sum_{i=1}^K r_i, \sum_{i=K+1}^M r_i\right\}\right] = \max\left\{E\left[\sum_{i=1}^K r_i\right], E\left[\sum_{i=K+1}^M r_i\right]\right\} \quad (9.7)$$

等式(9.7)中 E 表示期望值。等式(9.7)说明:所有不相关的具有相同分布的随机变量 R_i (任务的运行时间)的和的期望值的最大值,等于和的最大值的期望值。利用以上这两个假设,该模型就可以变为等式(9.2)所示的确定性模型,其结论也就显而易见了。

实际上等式(9.7)的假设有时不成立,但它所产生的结论还基本上正确。如果等式(9.7)中某一累加和的加数个数比别的任何一个多得多,那么几乎可以肯定它具有最大期望值,且该期望值等于等式(9.7)两边的值。如果两个或更多的累加和含有几乎相同数目的加数个数,且该数是等式中所有累加和的加数个数的最大值,那么就可能在等式左边选一项,右边选另一项,其累加和的值相当接近。所以,等式(9.7)不是完全正确但近似得相当好。

9.2.4 通信开销为线性函数的模型

在第一个模型中,假设每个任务与所有其它任务都有通信联系,这样,用于通信的额

外开销随处理机数目的增加以二次函数增加。这是每个任务都要向所有其它任务传递相同信息的情况,而这种结构的程序不太适合在多处处理机上运行。如果真有这种结构的程序,研究结果可以阐明能有多大加速比和多大代价。确实有许多程序适合在多处处理机上并行运行。我们需要知道这些程序潜在的性能和如何把它开发出来。

对于该模型,假设通信开销与处理机数目成正比例,而不是同分配的任务数成正比例。下面这种情况属于这种模型。例如一个任务要和其它所有任务通信,且通信的内容相同。这只需向每台处理机发送一次信息就可以了,当该信息到达某一台处理机后,在这一台处理机中的任务之间的信息传递就不必花费通信代价了。把任务分配给 N 台处理机时,总执行时间为:

$$\text{执行时间} = R\max(K_i) + CN \quad (9.8)$$

对于一个确定的 N 值来说,等式(9.8)的第一项和任务的分配有关,第二项与任务的分配无关。如果把任务平均地分配给所有处理机,那么第一项近似等于 RM/N ,使执行时间最小。当 N 增加时,第二项的增加甚至会超过第一项的减少。所以存在一个最大的 N 值,这时的性能最佳,这个 N 值是 R/C 的函数。

采用最佳分配方法会使第一项近似为 RM/N ,当处理机台数由 N 增加到 $N+1$ 时,执行时间的减少量近似为:

$$\begin{aligned} \text{执行时间减少量} &= RM\left(\frac{1}{N} - \frac{1}{N+1}\right) - C \\ &= \frac{RM}{N(N+1)} - C \end{aligned} \quad (9.9)$$

当下列等式成立时,执行时间减少量变为负值了,即不但没有减少,反而增加了。

$$R/C = N(N+1)/M \quad \text{或} \quad N = \sqrt{\frac{RM}{C}} \quad (9.10)$$

我们希望 M 项任务能在 N 台独立的处理机上快速执行,但本模型告诉我们,由于通信开销,并行度降为我们期望值的平方根。如果 R/C 值大一点,并行度可能会高些,所以这种情况下希望采用粗粒度,然而 R/C 值也要开平方,结果它的作用也削弱了。

当 N 达到等式(9.10)所示的临界值时,执行时间将不再减少。其实,在 N 达到那个临界值之前,我们每增加一台处理机所增加的开销和它所能获得的性能相比已经不合算了。这样,一个满足该模型且有 10 000 个任务的作业,可能在 100 台处理机上运行速度最快,但从经济效益角度看,可能在最多不超过 10 台处理机上运行最佳。

该模型与第一个模型的区别在于等式的第二项。第一个模型的第二项的值随常数 M 成平方增长,与 N 成反比例减少。如果把 N 项任务集中在一台处理机上执行,那么额外开销可以降低,所以和 N 有关。由于第一项和第二项都随着 N 的增加而减少,所以总执行时间随着 N 增加而减少。

在本模型中,第二项随着 N 线性增加,这是 N 超过临界值会使性能降低的原因。这两种模型告诉我们,额外开销需付出一定的代价。还告诉我们,在某种情况下限制并行性反而能获得高性能。

9.2.5 一个完全重叠通信的理想模型

迄今为止我们所讨论的模型都是比较悲观的,这是由于通信需要额外开销引起的。前面我们一直没有考虑将额外工作与有用的必须的计算过程重叠并行地进行。我们曾提到,在实际系统中一部分额外工作可以被有用的计算过程屏蔽掉,进而减少额外开销,但由于竞争、有限的通信宽度、同步等原因,有部分额外工作是不可能被屏蔽掉的。

下面我们给出一个理想的模型,它的额外工作可以与计算过程重叠进行而使额外开销趋于零。我们把等式(9.2)所示模型作简单的修改,使第二项的额外工作尽可能与第一项工作相重叠,这样,等式(9.2)就变为:

$$\text{执行时间} = \max \left\{ R \max(K_i), \frac{C}{2} \sum_i K_i (M - K_i) \right\} \quad (9.11)$$

对于两台处理机系统来说,等式(9.11)所描述的情况如图 9.2 所示。其中分段线性曲线表示第一项的开销,二次曲线表示第二项的开销。其交点是等式(9.11)所示最大函数的最小值。在这一点运行时间足够长,能完全屏蔽额外工作。

因为额外工作毕竟不能全部被屏蔽掉,所以该模型显然是理想的情况。尽管如此,我们可以利用该模型计算出关键点。对于两台处理机系统,图 9.2 中线性直线与二次曲线的交点是关键点。此交点在

$$R(M - K) = C(M - K)K \quad (9.12)$$

也即

$$K = R/C \quad (9.13)$$

处。其中 $1 \leq K \leq M/2$ 。如果将式(9.13)代入等式(9.11)中,计算时间等于 $R(M - R/C)$,加速比等于 $1/(1 - R/CM)$ 。由于等式(9.13)的 K 有一定范围,所以 R/C 也有一定范围,即 $1 \leq R/C \leq M/2$ 。当 R/C 在该范围内时,两台处理机系统的加速比在 1 与 2 之间,在 $R/C = M/2$ 时取得最大值,这和第一个模型的结论相同。

把任务平均分配给两台处理机时,即 $K = M/2$,则加速比最大。当 R/C 逐渐减为 1 时,加速比也逐渐减为 1,最优分配变得越来越不均衡了。所以,该模型同样依赖于 R/C 值。如果 R/C 足够大,全部或部分额外工作可以被计算过程屏蔽掉,所以性能更为乐观。

至于 N 台处理机系统,利用刚才的结论,额外工作重叠模型也就容易分析了。对于任何一个给定的能决定运行时间的 $K-i$ 最大值,平均分配任务能产生最少的通信时间。因此,最可能使通信完全被屏蔽掉的执行时间发生在下列等式成立时,

$$\frac{RM}{N} = \frac{CM^2}{2} \left(1 - \frac{1}{N} \right) \quad (9.14)$$

在 N 值很大的情况下,近似为:

$$\frac{R}{C} = \frac{NM}{2} \quad (9.15)$$

在这种具有最小总时间的情况下,处理机数目由下面一个关于 R/C 和 M 的函数式给出:

$$N = \frac{2R}{CM} \quad (9.16)$$

处理机数目的最佳选择与可提供的任务数成反比例。

随着可用并行度的增长,最好的策略是几乎不增加处理机数目。如果 N 值很小,那么我们不能忽略等式(9.14)中 $1/N$ 这一项,所得到的结论基本上没有改变。例如, $N=2$ 时,等式(9.14)在 $M/2=R/C$ 处取得最短时间,这与我们前面的结论是一致的。

该模型中处理机数目与并行度成反比例这一事实清楚地说明额外工作时间以比运行时间快 M 倍的速度增加。当运行时间小于通信时间时,总时间主要取决于通信所需额外开销时间,我们希望用计算时间来屏蔽额外工作时间就很难实现了。因此,如果想要加快速度,通信时间绝对不能比执行时间大。

9.2.6 一个具有多条通信链的模型

前面所讨论的模型都假设处理机并行地计算,使得各处理机的运行时间重叠。但是由参数 C 决定的额外操作须顺序地执行。如果额外开销仅仅是通信,那么这种模型只适用于处理机共用单个通信通道的系统。例如,所有处理机通过一条单总线或一条环路相连,或者所有处理机以不重叠存取方式访问同一个共享存储单元就属于这种情况。

我们可以采用增加通信链路和别的能消除额外开销瓶颈的系统结构方法。这样,参数 C 不再是一个常数而成为一个随 N 变化的函数了。下面我们考虑每个进程都需与别的进程进行通信的情况。最初估计的运算时间如等式(9.2)所示。如果允许通信链路随 N 的增加而增加,让每台处理机与别的任何一台处理机都有专门的链路相连,那么通信操作就能和进程本身重叠进行。然而,即使设置的链路数为 $O(N^2)$ 数量级,仍不可能支持超过 $O(N)$ 数量级的并发通信,因为一台处理机在某一时刻只能与一台处理机通信。这种情况下,我们可以将等式(9.2)中的第二项除以 N 后得到:

$$\text{执行时间} = R\max(K_i) + \frac{C}{2N} \sum_i K_i(M - K_i) \quad (9.17)$$

等式(9.17)假设一台处理机或者在进行计算,或者在传送信息,或者是空闲。因为同一时刻最多只有 N 台处理机在相互进行通信,信息传递的总时间与 N 成反比例。空闲时间是由于先完成任务的处理机必须等后完成任务的处理机引起的。

等式(9.2)中第一项随 N 的增大而减少,第二项却随着 N 线性地增加,这在前面已经讨论过。第一项可以通过均分任务给所有的处理机而达到最小,但同时由于第二项的增加而使总的执行时间并没有改善。

我们知道,对于任何 N 来说,尽可能平分任务可以使等式(9.17)达到最小,这样除一台处理机外,使所有其它处理机都分得最多的任务。按这种方式分配,等式(9.17)的执行时间可变为:

$$\text{执行时间} = \frac{RM}{N} + \frac{CM^2}{2N} \left(1 - \frac{1}{N}\right) \quad (9.18)$$

这种情况下,并行度越高越好,直到再增加处理机不能使运行时间减少为止。这个点发生在:

$$\text{执行时间减少量} = \frac{RM + \frac{CM^2}{2}}{N(N+1)} - \frac{\left(\frac{CM^2}{2}\right)(2N+1)}{[N(N+1)]^2} \quad (9.19)$$

提出因式 $M/N(N+1)$, 当 N 增至无穷大时, 等式(9.19)可简化为:

$$\text{执行时间减少量} = \left[R + \left(\frac{CM}{2} \right) \left(1 - \frac{2}{N} \right) \right] \left(\frac{M}{N(N+1)} \right) \quad (9.20)$$

在 $N > 2$ 时, 该式恒为正值, 所以除 N 值很小以外, 所有 N 都可能改善执行时间。

为了验证 N 台处理机是否比一台处理机效果更佳, 我们可以将等式(9.18)与单台处理机的运行时间 RM 比较。相等的情况是:

$$RM = \frac{RM}{N} + \left(\frac{CM^2}{2N} \right) \left(1 - \frac{1}{N} \right) \quad (9.21)$$

相等点在:

$$\frac{R}{C} = \frac{M}{2N} \quad (9.22)$$

在相等点参数 R/C 与 N 成反比例。因此, 在相等点, N 越大, 允许的粒度越小。

在相等点, N 台处理机总性能等价于一台处理机性能, 而其价格由于采用多处理机和通信系统却增加了。我们肯定不会在相等点使用并行系统。

从这个例子得到的启示是: 通过增加通信链路的带宽, 可使任务粒度比其它任何情况都要小。多条通信链路能够带来的速度优势在价格上是否合算, 很大程度上依赖于处理机与处理机之间所采用的通信技术。

总结本节所述的几个模型, 我们可以看到:

1. 多处理机系统结构所需的额外开销, 包括调度、对共享资源的竞争、同步、处理机之间通信等, 在串行机和向量机(或别的单指令流机)系统结构中是不存在的。
2. 当运行某个程序的处理机数目增加时, 用于计算的那部分时间将减少, 而额外开销时间却增加了。实际上, 额外开销的增加可能比处理机数目的线性增加更快。
3. R/C 比值表示当一个程序在某一特定系统结构上执行时, 程序执行时间(运行时间)与额外开销时间(通信时间)的比值。该比值越大, 越有利于计算过程。因为随着 R/C 比值的增加属于额外开销的时间相对减少了。如果将整个计算分成几大部分而不是很多小部分从而获得较大的 R/C 值, 那么, 并行程度将大为降低, 也就限制了在多处理机上能够获得的加速比。

很明显, 我们有点进退两难。一方面, R/C 要足够小使能并行执行的任务数目较多; 另一方面, R/C 要足够大以免额外开销太大。由于这个矛盾, 我们不能期望只靠简单地通过增加处理机数目制造出高速的多处理机系统。

处理机数目究竟多少才能使价格和性能都比较合理, 其实存在一个极大值, 这个值很大程度上依赖于机器的系统结构、基本技术(尤其是通信技术)和每一个具体应用问题的性质。

9.2.7 多处理机模型

多处理机给计算机系统结构设计者和算法设计者带来的是不同的问题。计算机系统结构设计者是要设计一个具有可接受的、尽可能高的 R/C 值的系统, 该系统能提供许多台使用效率很高的处理机。算法设计者面临的却是完全不同的问题。

假定一个由 N 台处理机组成的系统, 并给定了 R/C 值, 一个应用问题如何分配, 使

其在上述多处理机系统上运行能够最有效地利用资源呢？算法设计者必须将应用问题分配到多台处理机上去，并要选择一种粒度，使有效的并行计算与通信或其它额外开销达到某种平衡。

对于有些应用问题，最有效的方案可能并不是使用所能提供的全部处理机。有时候，用少量几台处理机就能完成这一工作，并且速度快成本低。实际上，我们是在确定用 1 头牛，还是 4 匹马，或 1 240 只鸡来犁田，哪一种方式更好。具有最高并行性的方案并不总是最快的方案。

很多人认为只要有工作可做，那么最好是有多多少台处理机就使用多少台。然而，本节讨论的一些模型告诉我们：当处理机数目增加时，计算速度有可能反而下降。所以最高的并行度，并不意味着最快的速度。此外，多处理机有时并不比本书前几章所讨论过的技术（例如流水线、向量处理机等）更能在可接受的价格条件下产生有效的速度。

例如 Cache 存储器能有效地提高主存速度，因为实际上仅有相对少的一部分存储器需要以 Cache 存储器速度工作。因此，采用 Cache 存储器技术能带来性能的提高，而付出的代价比较小。

同样，流水线计算机的性能改善与流水线的级数成正比例。在最好情况下，一个 N 级流水线可以获得 N 倍的加速比。但 N 倍的加速比并不需要 N 倍的硬件。

在上述两种情况中，需要重复的部件是导致系统中别的资源空闲的瓶颈，一旦瓶颈被打破，空闲资源便被利用起来了，那么可获得的总资源就会比在瓶颈状态下可获得的资源要多。

对于 Cache 存储器来说，其瓶颈是存储器，尤其是访问频繁的那部分存储区域。对于流水线来说，其瓶颈是某个计算段或某个关键寄存器。Cache 存储器技术是复制存储器，流水线技术是复制存储单元和算术部件，而多处理机要复制的部件是整个处理机，而不是处理机某些关键部件。而且，当我们增加处理机台数时，所获得的效益很可能不按比例增加。

所以，多处理机系统结构的设计远比我们前面讨论过的技术富有挑战性。不能简单地把 1 000 台处理机捆一块儿就想获得 1 000 倍的性能改善。实际上，即使在有利情况下，性能改善最多也只能达到 100~200 倍，而在不利情况下，性能改善可能只有 10 倍或更少。

另一方面，如果我们更深入地研究算法、设计技术和有关额外开销问题，那么有可能研制出高效的多处理机系统。上面分析中强调了随着处理机数目的增加有效性反而被限制的问题。目前一种 4~16 台处理机系统结构被认为是通用的系统，而 1K 或 64K 台处理机的系统结构几乎可以肯定仅适用于内在并行性很高的应用领域，其粒度也在系统能有效工作的范围之内。

在多处理机设计过程中，有效性是一个主要考虑的因素。一种低效地使用 $2N$ 台处理机的设计方案，在成本上是无法与使用 N 台处理机而有效性是 2 倍的设计方案相比拟的。

9.3 多处理机的 Cache 一致性

Cache 作为提高系统性能的一种常用手段在并行计算机系统中得到了很多的应用，

但是在并行处理机系统中的私有 Cache 会引起 Cache 中的内容相互之间以及与共享存储器之间互不相同的问题,这就是本节将要讨论的 Cache 一致性问题。

首先讨论 Cache 一致性问题的由来。然后讨论两种 Cache 一致性的解决方案:监听协议和基于目录的协议。

9.3.1 问题由来

加入了 Cache,整个存储系统的速度得到很大的提高。但是,相邻层之间和同一层之间可能会出现数据不一致的现象。出现数据不一致性问题的原因有三个:共享可写的数据、进程迁移和 I/O 传输。下面说明数据不一致现象的由来。

1. 共享可写数据引起的不一致性

只有当使用多个私有 Cache 时才会发生 Cache 不一致的问题,图 9.3 说明数据的不一致是怎样发生的:

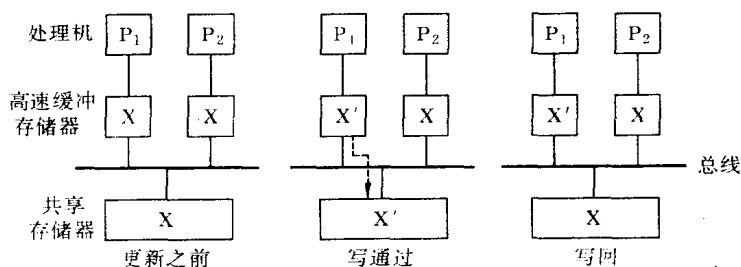


图 9.3 共享可写数据引起的 Cache 不一致性

图 9.3 以拥有两个处理机的系统为例,处理机带有各自的私有 Cache,并公用一个共享的主存储器。比如说,在 P_1 和 P_2 的本地高速缓冲存储器 C_1 和 C_2 中分别有内存某个数据 X 的拷贝,那么当 P_1 把 X 的值改变成 X' 之后,如果 P_1 采用写通过策略,即同时修改内存中的值,这时内存中的内容也变为 X' ,但是 P_2 处理机中的 C_2 的内容还是 X 。当 P_2 处理机要读 X 时,它读到的是 C_2 中的内容,这样就导致了 C_2 和内存中的内容的不一致性;如果 P_1 采用写回策略,即不立即修改内存中的值,这时内存中的内容还是 X ,当 P_2 处理机要读 X 时,读到的是 X 而不是 X' ,这样就导致了 C_1 和内存中的内容的不一致性。

2. 进程迁移引起的数据不一致性

与共享数据时类似,进程迁移也可能引起数据的不一致性,图 9.4 说明进程迁移是怎样造成不一致性的。

P_1 中有共享数据 X 的拷贝,而 P_2 中没有该共享数据, P_1 进程对 X 进行了修改,如果采用了写回策略,暂时没有对内存中的数据进行修改,由于某种原因该进程迁移到了 P_2 上运行,修改过的 X' 仍在 P_1 的 C_1 中。 P_2 运行时从内存中读取得到 X ,但是这个读到的 X 是“过时”的。

或者 P_1 和 P_2 中都有共享数据 X 的拷贝, P_2 修改了 X ,并采用写通过策略,所以同时内存中的 X 也修改成了 X' 。由于某种原因该进程迁移到 P_1 上,但这时 P_1 的 C_1 中仍然是 X 。

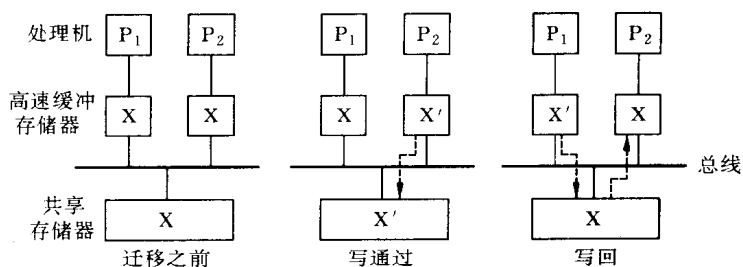


图 9.4 进程迁移引起的 Cache 不一致性

由于以上两种原因,都造成了 Cache 的不一致性。

3. I/O 传输造成的数据不一致性

图 9.5 说明 I/O 传输是怎样造成数据不一致性的。

绕过 Cache 的 I/O 操作也会产生不一致性问题。假设在 P_1 处理机和 P_2 处理机的本地高速缓存 C_1 和 C_2 中有某一个数据 X 的拷贝,当 I/O 处理机将一个新的数据 X' 写入内存中时,这样就导致了内存和 Cache 之间的数据不一致性;或者假设 C_1 和 C_2 中都有 X 的拷贝,但是 P_1 处理机在运行过程中修改了 X 的值,使之变为 X' ,又假设 P_1 采用写回策略,这样 P_1 中 C_1 的内容和内存中的内容是不一致的。这时候 I/O 处理机来了一个读 X 的请求,内存就直接把 X 的值传给了 I/O 部件,显然这个 X 是“过时”的。以上两种原因造成了 Cache 的不一致性。

一种解决 I/O 操作引起的不一致性问题的方法是把 I/O 处理机(IOP_1 和 IOP_2)分别连接到私有高速缓存 C_1 和 C_2 上,采用这种方法后,I/O 处理机就能和 CPU 共享高速缓存了,只要能够保证各 Cache 之间以及 Cache 和内存之间的数据一致性,就能够保证 I/O 操作的一致性。

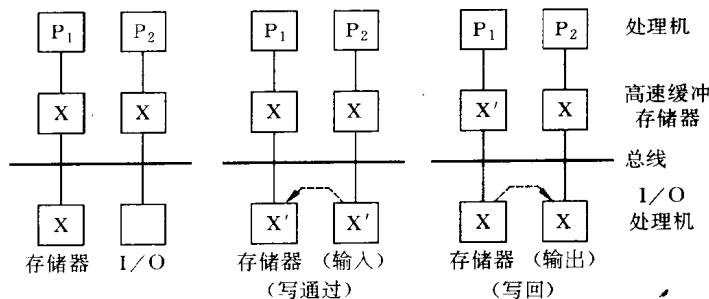


图 9.5 I/O 传输引起的 Cache 不一致性

为了解决 Cache 不一致性问题,提出了两类解决 Cache 不一致性问题的协议机制:监听协议和基于目录的协议。它们适合于不同的系统结构。下面就这两类协议进行讨论。

9.3.2 监听协议

根据系统结构的不同,解决 Cache 一致性问题的协议机制主要有两类。在采用基于总线互连结构的系统中,由于系统中每个处理机都能觉察到存储器系统正在进行的活

某个活动破坏了 Cache 的一致性时,Cache 控制器将采取相应的动作使有关的拷贝无效或更新。采用这种保持 Cache 一致性的协议称为监听协议。

但是,很多并行系统并不采用基于总线互连的结构,在这些系统中处理机无法对存储器系统的活动进行监听,监听协议也就不再适用。这类系统的 Cache 一致性问题可以采用基于目录的方法来解决。

1. 监听协议概述

在使用监听协议时,有两种来保持 Cache 一致性的方法:写无效(Write-Invalidate)策略和更新(Write-Update)策略。写无效策略是在本地 Cache 的数据块修改时使远程数据块都无效,写更新策略在本地 Cache 数据块修改时通过总线把新的数据块广播给含该数据块的所有其它 Cache。注意,写无效和写更新策略是保持 Cache 一致性的策略,与 Cache 采用写回(Write-Back)还是写通过(Write-Through)策略无关。如果 Cache 采用的是写通过策略,在使远程数据块无效或更新其它 Cache 的同时,还要同时修改共享存储器的内容。

图 9.6 说明 Write-Invalidate 策略和 Write-Update 策略的区别(以 Write-Through 方式为例)。

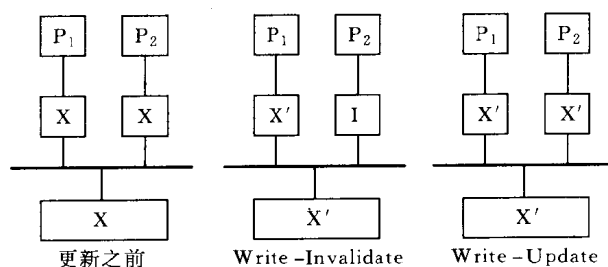


图 9.6 Write-Invalidate 策略和 Write-Update 策略

由于 Write-Update 策略在本地 Cache 修改时需要通过总线把修改过的数据块的内容广播给所有含该数据块的其它 Cache,增加了总线的负担,所以在一般的应用系统中,极少使用 Write-Update 策略。大部分系统使用 Write-Invalidate 策略。鉴于这种情况,在下面的讨论中我们就不再考虑 Write-Update 策略。

2. 采用 Write-Through 策略的 Cache

在使用 Write-Through 策略的 Cache 中,数据块有两种状态:有效和无效。有效表示该数据块内容正确,无效表示该数据块内容已“过时”或不在 Cache。这两种状态的转换如图 9.7 所示。

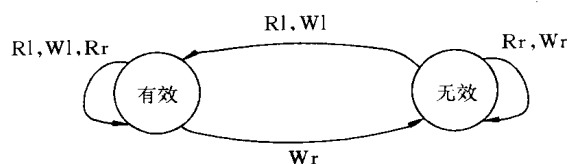


图 9.7 采用 Write-Through 策略的 Cache 状态图

图中, R_l 、 W_l 分别表示本地处理机对 Cache 某数据块的读和写操作, R_r 、 W_r 分别表示其它处理机对其 Cache 中存有相同内容的数据块的读和写操作。要注意:“有效”、“无效”指的是本地处理机相应数据块的状态,并非整个 Cache 的状态,但为了叙述方便,下面有些部分把“Cache 某个数据块的状态”简述为“Cache 的状态”。

从图中可以看出,在 Cache 无效时,其它处理机的任何操作都不会影响本地 Cache 的无效状态。只有在本地处理机读或写了数据块中的某个数据,即对 Cache 执行了 Read 或 Write 命令时,该数据块的状态才会成为“有效”。同样,本地处理机对有效数据块进行的读、写操作,以及其它处理机对存有相同内容的数据块的读操作也都不会影响该数据块的有效状态。只有在其它处理机执行了对存有相同内容的数据块的写操作时,该数据块的状态才会成为“无效”。

3. 采用 Write-Back 策略的 Cache

与采用 Write-Through 策略的 Cache 相比,采用 Write-Back 策略的 Cache 的状态要复杂一些。在采用 Write-Through 策略的 Cache 中的有效状态在这里被进一步细分为两种:读-写(read-write)状态和只读(read-only)状态。只读状态表示整个系统中不止一个数据块拷贝是正确的,例如一个在 Cache 中,另一个在存储器中。读写状态表示数据块至少被修改过一次,存储器中相应数据块还没有被修改,即在整个系统中只有一个数据块拷贝是正确的。图 9.8 说明使用 Write-Back 策略的 Cache 中数据块状态的转换:

图中 R_l 、 R_r 、 W_l 、 W_r 的含意和图 9.7 相同。

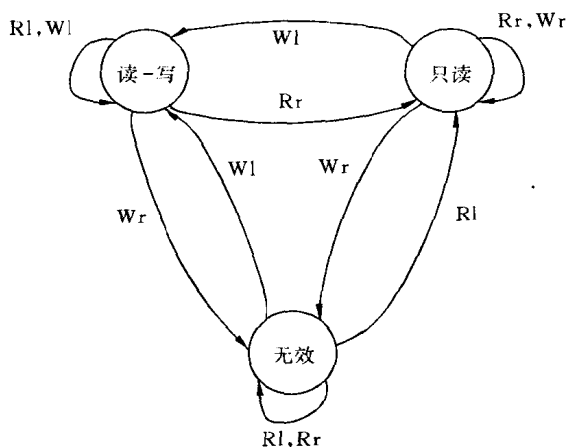


图 9.8 采用 Write-Back 策略的 Cache 状态图

从状态图中可以看出,在系统中不可能同时存在多于一个的存有相同数据的读-写状态的数据块。如果一个 Cache 中的某一个数据块处于读-写状态,则其它的 Cache 中就不可能存在有效(读-写或只读)的存有相同数据的数据块。对于只读的数据块,本地的和远程的(其它处理机的)读操作都是安全的(不改变当前的状态)。本地的写操作将引起其状态转移为读写状态,远程的写操作将使之变为无效。对于读-写状态的数据块,本地的读、写操作都是安全的,而远程的读操作将引起写回动作,并且将数据块正确的内容传递给读操作的远程处理机的 Cache,然后两个 Cache 的状态都转移至只读状态。远程的写操作的

情况与之类似,只是在传递数据块之后,远程的处理机将对数据块进行写操作,远程处理机 Cache 的状态转移至读-写状态,而本地的 Cache 将转移至无效状态。

对于从无效状态向读-写和只读状态的转移则简单地取决于引起转移的操作是本地读还是本地写(其它操作不会引起从无效状态向其它状态的转移),如果操作是本地读,则转移至只读状态;如果操作是本地写,则转移至读-写状态,同时使其它 Cache 中相应数据块的状态转移为无效状态。

4. 写一次(write-once) 协议

Write-Back 和 Write-Through 两种策略都有其弱点,所以,1983 年 James Goodman 提出了 Write-Once 的 Cache 一致性协议。它适用于基于总线的多处理机系统,这种方法把 Write-Back 和 Write-Through 两种策略的优点结合在一起,其特点是:为了减少总线流量,Cache 的第一次写采取 Write-Through 策略,而后的写则采取 Write-Back 策略,此时,整个系统只有一份正确的拷贝。

为了区分是否第一次写,协议把“读-写”状态分为两个状态:“保留(reserved)”和“重写(dirty)”,现在 Cache 中数据块的状态就成了 4 种:

- “有效”(valid, 相当于 Write-Back 里的“只读”):从存储器读入的并与存储器拷贝一致的 Cache 数据块。
- “无效”(invalid):在 Cache 中找不到或 Cache 中的数据块内容已“过时”。
- “保留”(reserved):数据从存储器读入 Cache 后只被写过一次,Cache 中的拷贝与存储器中的拷贝是一致的,并且它是正确的拷贝。
- “重写”(dirty):Cache 中的数据块不只一次被写过,它是唯一正确的数据块,此时存储器中的数据块也不是正确的数据块。

图 9.9 是 Write-Once 协议的状态图。

图 9.9 中, Rl、Rr、Wl、Wr 分别表示本地读、远程读、本地写、远程写。

下面具体地描述 Write-Once 协议是怎样保持 Cache 一致性的。我们就 CPU 对 Cache 的不同命令来叙述 Cache 的状态变换。

- CPU 读 Cache:当处理机启动总线读操作时,有两种可能性。一种可能性是在 Cache 中存在有效的(有效、保留或重写的)数据块时,CPU 直接读取数据,Cache 状态不变。另一种可能性就是 Cache 中不存在有效的数据块,即数据块处于无效状态。此时将触发读缺失(read-miss)事件,系统设法将有效的数据块调入 Cache。具体的过程如下:首先判断系统中是否存在处于有效、保留或重写状态的相应数据块,如果存在,则将其调入本地 Cache;在相应数据块处于重写状态时,还要同时禁止存储器操作。如果系统中不存在处于有效、保留或重写状态的相应数据块,则说明存储器中的数据块是正确的拷贝(也是唯一的拷贝),这时直接从存储器中读入

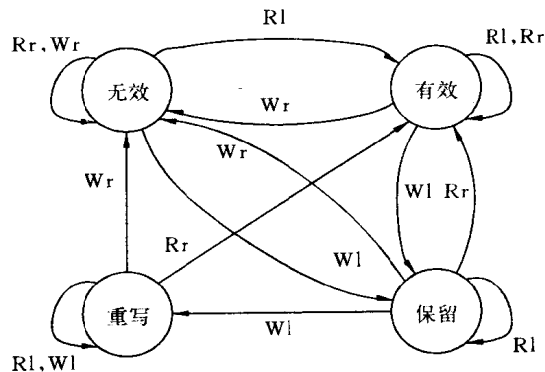


图 9.9 Write-Once 协议的状态图

就可以了。无论哪种情况,读入后 Cache 中的相应数据块将进入“有效”状态。

- CPU 写 Cache:处理机启动总线写操作时,与总线读操作相似,也有两种可能。或者命中,或者不命中。当写命中时,将引起 Cache 状态的转移。具体地说,当 Cache 状态处于“有效”状态时,将采用 Write-Through 策略,把写入 Cache 的内容同时写入存储器,并将 Cache 的状态转移为“保留”,同时将其它 Cache 的相应数据块状态置为“无效”;当 Cache 处于“保留”或“重写”态时,使用 Write-Back 策略,Cache 的状态转移至“重写”态(此时其它的存有相同内容的 Cache 一定是处于“无效”态,所以这些 Cache 无需再进行状态转移)。当写不命中时,触发写缺失(write-miss)事件,系统首先将正确的数据块调入 Cache(调入的方法同读缺失),然后写 Cache,因为是第一次写,所以使用 Write-Through 策略,同时写存储器。此时状态是这样转移的:将本地 Cache 的状态置为“保留”,同时将其它 Cache 的相应数据块状态置为“无效”。

Write-Once 协议也有其不利的一面。由于当主存储器的内容无效时。读缺失引起的总线读操作必须禁止主存储器的操作(以免造成总线冲突),而大多数总线不支持这种操作。Write-Once 协议最大的好处就是由于第一次写和以后的各次写操作分别采用了 Write-Through 和 Write-Back 策略,减少了无效操作,提高了总线的效率。

除了上面提到的这些协议之外,监听总线协议还包括 Futurebus+ 等协议。这里就不一一说明了。下面我们将讨论基于目录的协议。

9.3.3 基于目录的协议

当某台处理机采用 Write-Invalid 协议正在更新一个变量同时其它处理机也试图读该变量的时候,则会发生读缺失并可能导致总线流量大大增加。另外,Write-Update 协议更新远程 Cache 中的数据,但这些数据可能永远也不会使用。这些问题严重限制了采用总线来构造大型多处理机系统。

由于在多级网络上实现广播功能的代价很大,所以监听协议就无法使用,把使其它 Cache 数据块无效的一致性命令只发给存放相应数据块的 Cache 是一个很好的解决办法。这样就产生了基于目录的协议。

1. 目录结构及使用

在多级网络中,用 Cache 目录存放有关 Cache 拷贝驻留在哪里的信息,从而支持 Cache 一致性。各种基于目录协议的不同之处主要在于目录如何维护和存放什么信息。Cache 目录的方案有集中式和分布式两种,它们的主要区别就在于其 Cache 目录(即 Cache 地址表)的存放形式。无论 Cache 目录采用集中方式还是分布方式来存放,其内容都是大量的指针,用以指明块拷贝的地址。每个目录项还有一个重写位,用以指明是否有一个 Cache 允许把有关的数据写入。

不同目录协议的区别在于目录的结构不同。根据目录的结构,可以把目录协议分成三类:全映射(full-map)目录、有限(limited)目录和链式(chained)目录。全映射目录存放与全局存储器中每个块有关的数据。这样,系统中的每个 Cache 可以同时存储任何数据块的拷贝。而在有限目录中,无论系统的规模多大,它的每个目录项的指针数是固定的,所以每个数据块能够装入 Cache 的数是有限的。链式目录的特点是把目录分布到全部的 Cache

中,其余部分与全映射相同。

目录的使用实际上很简单:在一个 CPU 对 Cache 进行了写操作的时候,系统根据 Cache 目录的内容将所有其它存有相同内容的 Cache 拷贝无效,并置重写位为“重写”。在 CPU 对 Cache 进行读操作的情况下,如果重写未置位,则说明该内容未经重写,此时若 Cache 读缺失,则从主存储器中或拥有正确内容的 Cache 中读入块并修改目录就可以了。如果读命中,那么直接读就可以了。

2. 全映射目录

用全映射目录协议实现的目录项中有 N 个处理机位和一个重写位。处理机位表示相应处理机对应的 Cache 块的状态(存在或不存在)。如果重写位为“1”而且有一个且只有一个处理机位为“1”,则意味着该处理机可以对该块进行写操作。

Cache 的每个数据块有两个状态位。一位表示数据块是否有效,另一位表示有效块是否允许写。Cache 一致性协议必须保证目录的状态位与 Cache 数据块的状态位一致。

图 9.10 是全映射目录的三种状态。

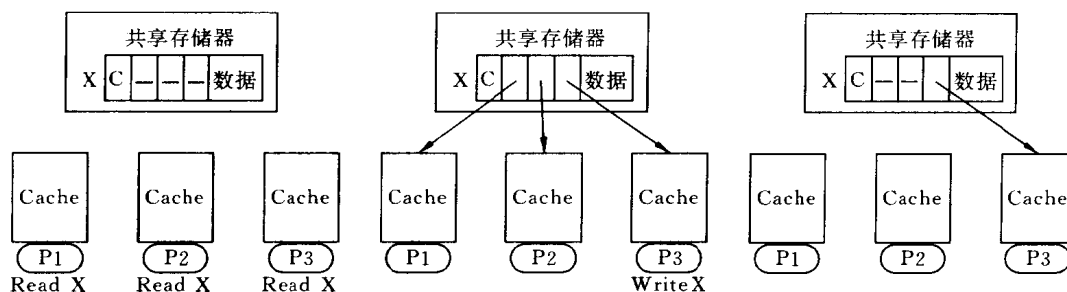


图 9.10 全映射目录的三种状态

图 9.10 是一个拥有三个 CPU 的并行系统的 Cache 状态。第一个图表示全系统中所有 Cache 都没有单元 X 的拷贝。当三个处理机都对 X 有过读请求之后,目录就进入第二种状态。三个处理机位都被置“1”,表示三个 Cache 中都有 X 的拷贝。第三种状态表示 P_3 处理机获得了对 X 的写权力之后的状态。

从第二种状态转移至第三种状态的过程比较复杂。可以用下面的步骤说明:

当 P_3 要求写单元 X 时:

- (1) C_3 发现包含 X 单元的块是有效的,但是不允许写。
- (2) C_3 向包含 X 单元的存储器模块发写请求,并暂停 P_3 的工作。
- (3) 该存储器模块发无效请求至 C_1 和 C_2 。
- (4) C_1 和 C_2 接到无效请求后,将对应块置为无效态,并发回答信号给存储器模块。
- (5) 存储器模块接到 C_1 和 C_2 的回答信号后,置重写位为“1”,清除指向 C_1 和 C_2 的指针,发允许写信号到 C_3 。
- (6) C_3 接到允许写信号,更新 Cache 状态,激活 P_3 。

至此,全部过程结束, P_3 就可以写 X 单元了。在 P_3 完成写操作之前存储器系统一直等待回答信号。

全映射目录协议的效率比较高,但是其开销与处理机数目的平方成正比(目录的项数

与处理机数目成正比,项的大小又与处理机数目成正比,所以其开销等于目录的项数乘以项的大小,即与处理机数目的平方成正比)。由于其过多的存储器开销,所以不具有扩展性。

3. 有限目录

有限目录协议可以解决目录过大的问题。通过对一个数据块在 Cache 中的拷贝数做一定的限制,则目录的大小将与处理机数目(存储器大小)与处理机数目对 2 的对数之积 ($O(N\log_2 N)$) 成正比(而不像全映射目录中与其平方成正比)。

有限目录协议的状态与全映射目录协议十分相似。唯一的区别就在于多于允许数目的 Cache 同时要求某一个数据的拷贝的时候。图 9.11 表明这时的做法。

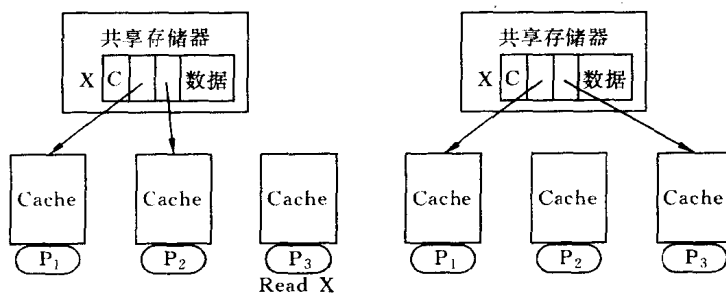


图 9.11 有限目录的驱逐

当 P_1 和 P_2 的 Cache 中都有 X 的拷贝时,若 P_3 请求 X 的拷贝,则存储器系统必须在 C_1 和 C_2 中选择一个使之无效。这种指针替换过程称为驱逐。由于多处理机系统中的处理机具有局部性(即在任何给定的时间间隔内,只有一小部分的处理机访问某个给定的存储器数据),所以有限目录足以应付这个小的处理机组了。

由于存在驱逐的问题,像 Cache 的替换需要有 Cache 替换策略以确定哪部分 Cache 内容需要被替换掉一样,有限目录的驱逐也需要一个驱逐策略来决定哪个指针将被驱逐。这个驱逐策略的好坏对系统的性能将具有很大的影响。幸运的是,驱逐策略与 Cache 替换策略在很多方面是相同的,所以在 Cache 替换策略方面的许多研究成果可以被直接用在驱逐策略的设计中。

由于在有限目录中,目录指针需要对处理机的二进制表示进行编码,所以每个指针占 $\log_2 N$ 位存储器,其中 N 是处理机的数目。这就是目录所占的空间与 $N\log_2 N$ 成正比的原因。

4. 链式目录

链式目录的优点在于既不限制共享数据块的拷贝数目,又保持了可扩展性。其主要方法是通过维护一个目录指针链来跟踪共享数据拷贝。

链式目录有两种实现方法。其中比较简单的一种是单链法,图 9.12 是单链法的工作原理。

假设 C_1 没有单元 X 的共享拷贝,如果处理机 P_1 要读单元 X,则存储器送一份拷贝给 C_1 ,同时送给 C_1 一个链结束指针(CT),存储器也保存一个指向 C_1 的指针。之后,当处理机 P_2 读单元 X 时,存储器送一份拷贝给 C_2 ,同时送给 C_2 一个指向 C_1 的指针。存储器则

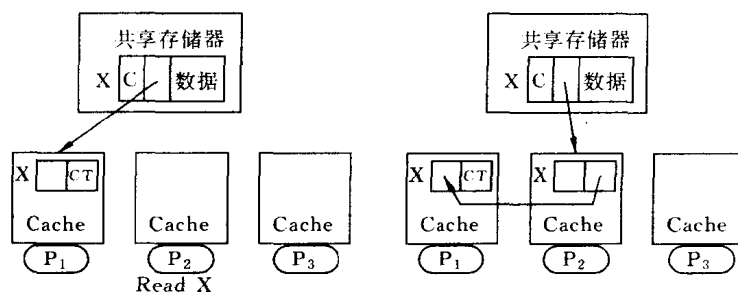


图 9.12 链式目录

保存一个指向 C_2 的指针。当某一个处理机需要写 X 时,它必须沿这个目录链发送一个数据无效信息。在有链结束指针的处理机没有回答无效信号之前,存储器模块不给该处理机写允许权。

与另外两种协议不同,链式目录协议存在一个新的问题。当 Cache 的数据块需要替换时,需要把该 Cache 从目录链中“卸”下来。这里就涉及链中某一项的删除问题。有两种解决这个问题的办法:

(1) 沿着链发送一个消息,使得 C_{i+1} 的指针指向 C_{i-1} ,把 C_i 从链中去掉。这时 C_i 中存放别的数据块了。

(2) 使 C_i 及在链中位于其后的所有 Cache 中的单元 X 无效。

解决替换问题的另一种方法是使用双向链。即每个 Cache 目录项都有向前链和向后链两个指针。这样在执行替换时就不再需要遍历整个链。虽然双向链优化了替换操作,但是其指针增加了一倍,占用了更多的资源,一致性协议也更复杂了。

链式目录的复杂程度超过了前两种目录。但是链式目录具有前两种目录所不具有的一个重要特性:可扩展性。其指针的大小以处理机数目的对数关系增长,Cache 的每个数据块的指针数目与处理机数目无关。

9.4 多处理机实例

多处理机系统主要有四大类。第一类是多向量处理系统,以 CRAY YMP-90、NEC SX-3 和 FUJITSU VP-2000 等为代表;第二类是基于共享存储的多处理机系统,如 SGI Challenge 和 Sun SparcCenter 2000;第三类是基于分布存储的大规模并行处理系统(MPP),比如 Intel Paragon、CM-5、Cray T3D 等;第四类是机群系统。

9.4.1 MPP

大规模并行处理(massively parallel processing, MPP)系统的定义随着时间推移在不断地变化。按照当前的标准,具有几百或几千台处理机的任何机器都是大规模并行处理系统。显然,随着计算机技术的快速发展,对并行度的要求会愈来愈高。

科学计算中的重大挑战性课题都离不开计算机的支持,例如:磁记录工业要靠用计算机来研究静磁和交互感应以降低高密度磁盘金属薄膜镀层的噪音;用计算机辅助设计合

适的药物,用以治疗癌症和后天免疫缺乏综合症。现在,用高性能计算机已经发现一种新的试剂,能抑制人工免疫缺乏病毒蛋白酶的作用。在超级计算机上研究计算流体动力学将辅助设计出高速民航飞机。通过化学动力学的计算可设计出较好的发动机模型,从而提高燃料的燃烧效率。对许多由酶催化控制的生物过程可用计算机来设计用于化学反应的催化剂。大规模并行量子模型要进行大量模拟以减少催化剂设计以及它们特性优化所需的时间。没有超级计算能力的 MPP 系统是不可能对海洋精确建模的。臭氧耗损的研究也要求用计算机分析其复杂的化学和动力学机制。海洋活动和臭氧耗损都将影响全局天气预报。其它需要计算支持的重要领域有实时医疗诊断的数字解析、通过计算建模研究减少大气污染、计算生物学家设计蛋白质结构、图象和图象理解以及科研与教学相结合的技术。某些重大挑战性课题的计算需求如图 9.13 所示。这张图列出了支持科学模拟、先进计算机辅助设计和大型数据库与信息检索操作的实时处理等所需要的处理速度和存储器规模的量级。重大挑战性课题要求能提供 1 Teraflops 计算能力、1 Terabyte 主存储器和 1 Terabyte/s I/O 带宽的计算机,这就称为 3T 性能目标。目前,性能最好的计算机与重大挑战性课题的这些要求相比,其速度还需提高 100 至 10 000 倍,存储容量太小,I/O 带宽还太窄。

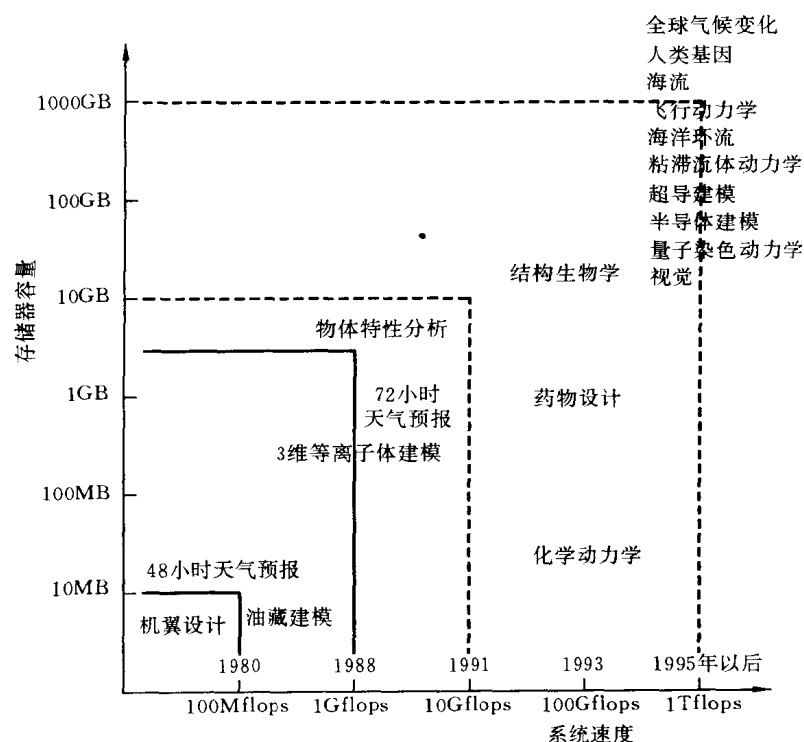


图 9.13 重大挑战性课题的需求

MPP 系统最重要的特点是进行大规模并行处理。在 MPP 系统中,用的是 VLSI 硅片、砷化镓技术、高密度组装和光技术,采用可扩展技术、共享虚拟存储技术、容许时延技术、多线程技术的系统结构。

MPP 系统采用分布存储方式可以使系统容易扩充,但由于每个处理机不能直接访问非本地存储器,只能使用消息传递方法来解决这个问题,这使得编程困难并且通信开销增加。虚拟共享存储器(shared virtual memory),又称共享分布存储器(distributed shared memory)可以改善这一状况。这是在基于分布存储器的多处理机上,实现物理上分布但逻辑上共享的存储系统。其基本思想是,将物理上分散的各个处理机所用的局部存储器,在逻辑上加以统一编址,形成一个统一的虚拟地址空间来实现存储器的共享。每个处理机可以访问全局存储器的任一位置,用户可以把它当成全局共享存储系统。这样,用户以前在全局共享多处理机系统上编写的程序就可以不加修改地在虚拟共享存储器系统上运行,这给软件的移植带来了方便,同时解决了难于对复杂数据结构进行传递和难于进行进程迁程的问题。虚拟共享存储器系统的优点有:编程容易,系统结构灵活,可扩充性好,有较好的软件移植性。已有的研究表明,在虚拟共享存储器系统上编制的程序比用消息传递方式编制的程序效率要高,这是因为:首先,在虚拟共享存储器系统上,数据都是以块的方式进行传送,如果一个程序,具有较高的局部性,则当把一个数据块传送到一个结点后,该结点对它的访问就成为本地访问,而消息传递方式的每次访问都需要通信。第二,许多并行应用程序都是分阶段执行的,每次执行前,都有一个数据交换阶段,其时间受通信限制。在虚拟共享存储器系统中,数据只有用到的时候才传送,取消了数据交换阶段,把通信时间加以分散,提高了并行性。最后,虚拟共享存储器系统提供的虚存空间比单个结点的存储空间大得多,减少了换页操作。因此,虚拟共享存储器系统受到了越来越广泛的重视。目前,实现虚拟共享存储器系统的途径有三种:(1)硬件实现。将传统的 Cache 技术扩展应用到松耦合分布式存储多处理机。这种途径需要在现有的松耦合分布式存储多处理机上,增加专用部件以取得高效的实现。(2)操作系统和库实现。通过虚拟存储管理机制取得共享和一致性。这种途径可在现有松耦合分布式存储多处理机上不增加任何专用部件实现。(3)编译实现。自动将共享访问转换成同步和一致原语。它要求用户显式控制全局数据,当传送大量数据时或试图进行迁移时极其复杂。现有的虚拟共享存储器系统大多数采用第一途径、第二途径或这两个途径结合实现的。

具有重大挑战性的应用问题将推动现在和未来的 MPP 系统朝 3T 性能的目标前进。Connection Machine 的 CM-5 是 Thinking Machines 公司为实现这一目标推出的系统。下面介绍 CM-5 的系统结构。

1. 同步的 MIMD 机器

按照惯例,超级计算机的程序员往往要在 MIMD 计算机和 SIMD 计算机之间作出选择。MIMD 机器的优点是善于处理独立的转移,但是同步和通信方面存在不少问题。相比之下,SIMD 机器的同步和通信功能很强,但转移处理的能力较差。CM-5 采用通用的系统结构,把 SIMD 和 MIMD 机器的优点结合在一起。CM-5 设计成同步的 MIMD 结构可同时支持两种并行计算方式。

CM-5 的系统结构如图 9.14 所示。机器包含 32 到 16 384 个处理器结点。每个结点有一台 32MHz 的 SPARC 处理机,32 兆字节的存储器和可以执行 64 位浮点和整数操作,速度为 128Mflops 的向量处理部件。

系统采用若干台 SUN 公司的工作站计算机作为控制处理机。控制处理机的数目根

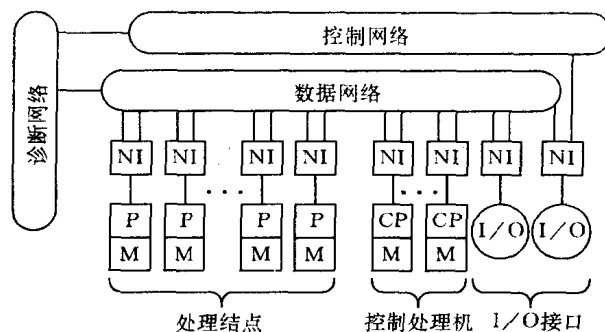


图 9.14 CM-5 的系统结构

据配置的不同可从 1 台到几十台变化。每个控制处理机根据需要配有存储器和磁盘。

系统的输入和输出通过高频宽的 I/O 接口与图形设备、海量辅助存储器以及高性能网络相连。与控制处理机相连的以太网提供低速 I/O 功能。配置在最大时的占地面积为 $30\text{m} \times 30\text{m}$ ，峰值速度可望超过 1Tflops。

CM-5 系统有三个网络：数据网络、控制网络 and 诊断网络。数据网络提供处理结点之间高性能点对点的数据通信。控制网络提供协同操作，包括广播、同步、扫描和系统管理功能。

诊断网络允许从“后门”访问所有的系统硬件以便测试系统完整性，检测和隔离错误。数据网络和控制网络通过网络接口与处理结点、控制处理机和 I/O 通道相连。

CM-5 系统结构是一种通用结构，因为它对大型复杂问题的数据并行处理能获得比较理想的结果。数据并行可用 SIMD 模式、多 SIMD 模式或者同步的 MIMD 模式实现。

数据网络和控制网络设计成要有很好的可扩展性，使得机器的规模可以根据所能承受的价格加以选择，而不受系统结构和工程的限制。换句话说，网络设计成与处理机的类型无关。当新技术可付诸实用时，可以很容易把它们用到原来的系统中去。网络接口则设计成能提供抽象的网络形式。

系统可以划分成一个或多个分区供用户使用。每个分区由一台控制处理机、一组处理结点以及数据和控制网络的专用部分组成。系统管理程序负责资源的划分。分配给每个分区的控制处理机起分区管理器的作用。每个用户进程在一个分区上运行，但可能要与其它分区的进程交换数据。由于每个分区都利用 UNIX 的分时和保密功能，因此每个分区允许多个用户来访问，保证不会发生冲突和干扰。

访问系统功能分为有特权和无特权两类。有特权的系统功能包括对数据和控制网络的访问。用户代码可以直接执行这些访问而不必使用系统调用。这样，操作系统内核不必为用户任务的网络通信增加开销。访问诊断网络、共享 I/O 资源以及其它分区也是有特权的访问，但必须通过系统调用实现。

CM-5 的一些控制处理机用来管理 I/O 设备和接口。这种结构允许任何分区中的进程可以访问任何 I/O 设备，并且保证对一个设备的访问不会妨碍对其它设备的访问。系统的操作从功能上可分为面向用户的分区管理、基于系统调用的 I/O 服务、数据和控制网络的动态控制以及系统管理和诊断。

两个网络负责把控制处理机中的用户代码装入处理结点,传送 I/O 请求,传送控制处理机之间的所有各种类型的信息,传送结点和 I/O 设备之间数据,所有这些可以在一个分区中也可以在不同的分区之间进行。随着处理结点数目或控制分区数目的增加,I/O 的能力也扩大了。CM-5 有许多特点,如硬件模块化、分布式控制、时延容忍和用户抽象,所有这些对于实现可扩展计算都是十分必需的。

2. CM-5 网络系统结构

数据网络是以 Leiserson 提出的胖树概念为基础设计的。胖树很像一棵真正的树,叶子越多时,它将变得越粗。处理结点、控制处理机和 I/O 通道都位于胖树的叶子上。树的内部结点都是开关。与一般的二进制树不同,从叶子到根,胖树的通道能力也随着增加。

利用胖树的层次结构性质可以划分一棵专用的子树给每个用户分区,但它不能作为与其它任何分区之间的信息流量的接口。CM-5 数据网络实际上是用 4 元胖树实现的,如图 9.15 所示。每个内部开关结点由 n 个寻径器芯片组成。每个寻径器芯片与 4 个子芯片和 2 个或 4 个父芯片相连。

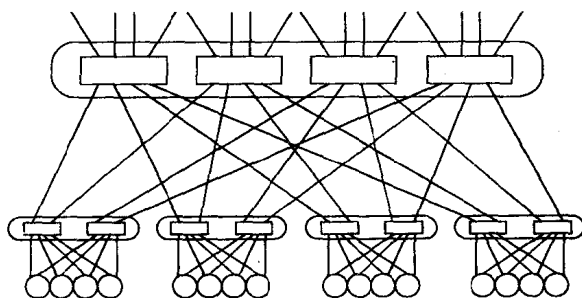


图 9.15 用 4 元胖树实现的 CM-5 数据网络

为了实现划分,可以分配不同的子树处理不同分区的作业,子树的大小可以随不同分区的要求而不同。可以将 I/O 通道分配给另一棵子树,但它不能为任何用户分区专用。这样,I/O 子树就作为共享的系统资源来访问。数据网络的功能在很多方面与层次的系统总线相似。不同之处是在划分的子树之间没有接口。所有叶子结点的物理地址是唯一的。为了把消息从一台处理机传送到另一台处理机,首先沿树将消息向上传送到离两台处理机最近的公共祖先,然后向下传送到目的处理机。

在用 4 元胖树实现的数据网络中,每个连接有一条与另一个芯片相连的链,这条链每一方向的频宽为 20 兆字节/秒。在树的每一层通过选择 2 条或 4 条父链可以调节胖树结点之间的频宽。每条链都有流控制机制。

每台处理机与数据网络有两条连接通路,每个叶子结点的输入和输出的频宽为 40 兆字节/秒。在前面两层,每个寻径器芯片只使用两条父连接通路与高一层相连,因此 16 个叶子结点的子树的总输出频宽为 160 兆字节/秒。所有高于第二层的寻径器芯片有 4 条父连接通路,因此对结点数为 2K 个的系统来说,从系统一半到另一半的每个方向的总频宽为 10G 字节/秒。

当 CM-5 的最大配置达 16384 个结点时,它的频宽仍能线性地增加。在规模较大的机器中,可以采用传输线技术在长线上以流水线方式传输数据位,这就克服了由于线延迟而

造成的频宽限制。

当一个消息沿树向上传送时,使用哪条父连接通路则有几种选择。我们可以从没有被其它消息占用的那些链中伪随机地选择一条,在到达离源结点和目的结点最近公共祖先后,则可通过一条可用的链路将消息向下送到目的结点。对每一层的伪随机选择能自动地平衡网络的负载,并且能避免病态消息所引起的过分拥挤。

数据网络芯片的时钟频率为 40MHz。前面两层通过底板相连。高层则通过电缆相连,电缆长度为 9 尺或 26 尺。电缆能以超过 90% 的光速可靠地传送信号。消息寻径以虫蚀技术为基础。

可以把故障处理机结点或连接链脱离系统而隔离开来。这样,当要对已脱离系统的这部分进行服务或测试时,系统仍然可以工作。数据网络从输入到输出没有环路。如果网络能保证最终接收或发送输入的信息,并且源处理机保证在它们成功地发送消息后把所有消息从网络中删去,那么就可以避免产生死锁。

控制网络的系统结构是一棵完全二叉树。所有的系统部件都在叶子上。可以给每个用户分区分配给一棵网络子树。处理结点位于子树的叶子处,控制处理机位于分区另外的叶子上。控制处理机执行代码中的标量部分,而处理结点执行数据并行部分。

与数据网络传送可变长度的消息不同,控制网络包的长度固定为 64 位。控制网络主要有三类操作:广播、组合和全局操作。这些操作实现处理机之间的通信。网络接口中独立的 FIFO 分别与每类控制操作相对应。

控制网络还提供能有效地执行数据并行代码的机构,并且支持以 MIMD 方式执行通用应用程序。二叉树系统结构使得控制网络比用胖树的数据网络实现起来要简单得多。控制网络还有切换能力,可以绕过故障并采用脱机寻径策略把任何一台控制处理机与任何一个用户分区相连。

为了使系统升级时有很好的可用性,十分需要这种诊断网络。内部可测试性是用基于扫描的诊断技术实现的。为了简化寻址,也将这种网络构造成一棵(不必是平衡的)二叉树。根结点上有一台或多台诊断处理机。叶子是由一块板或底板构成的物理系统盒子。从根到每个被测试盒子的路径是唯一的。

诊断网络允许一组盒子按“超立方体编址”模式进行编址。一个专用的诊断接口用来对系统内部支持 JTAG (join test action group) 标准的所有 CM-5 芯片以及所有的网络进行总体测试。它可以扫描访问支持 JTAG 标准的所有芯片以及可编程访问特定的非 JTAG 芯片。网络本身是完全可测试和可诊断的。它能够发现和排除机器的故障或掉电部件。

3. 控制处理机和处理结点

控制处理机如图 9.16 所示,基本的控制处理机由 RISC 微处理器(CPU)、存储器子系统、带本地磁盘和以太网连接的 I/O 以及 CM-5 网络接口

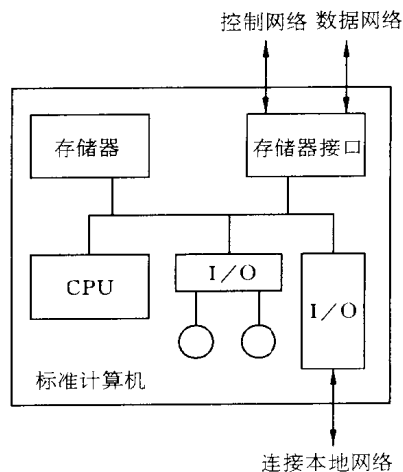


图 9.16 CM-5 的控制处理机

组成。它相当于一个流行的标准工作站类计算机系统。网络接口通过控制网络和数据网络将处理机与系统的其它部分相连。

每台控制处理机运行 CMOST,它是基于 UNIX 的操作系统,负责 CM-5 并行处理资源的管理。一部分控制处理机用来管理用户分区中的计算资源,其它的则用于管理 I/O 资源。控制处理机专门执行管理功能而不是计算功能,因此它不需要高性能的运算加速器。相反,在控制处理机中增加 I/O 连接则更为有用。

图 9.17 给出了处理结点的基本结构。它是一台有存储器子系统的基于 SPARC 的微处理机。存储器子系统由存储器控制器和 8、16 或 32 兆字节的 DRAM 存储器组成。内部总线宽度为 64 位。

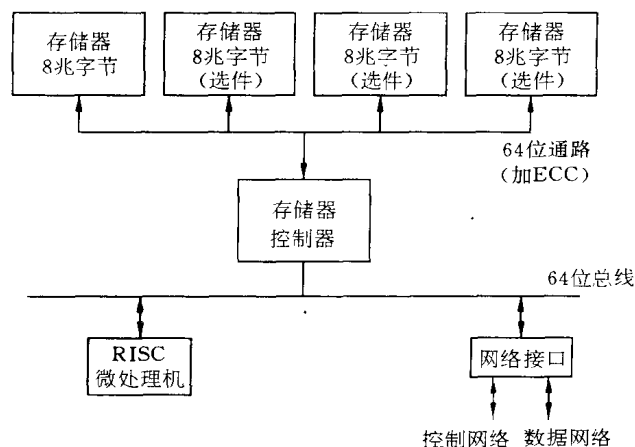


图 9.17 CM-5 的处理结点

选择 SPARC 处理器是因为它有多窗口特征,易于实现快速上下文切换。这对在不同时间不同用户分区能够动态地使用处理结点来说非常重要。网络接口通过控制网络和数据网络将结点与系统的其它部分相连。可选的硬件运算加速器可以提高微处理器的能力。

向量部件如图 9.18 所示。向量部件作为选件可以加在存储体和系统总线之间。每个向量部件有一条宽度为 72 位的专用通路,与附属的存储体相连。每个向量部件的存储器

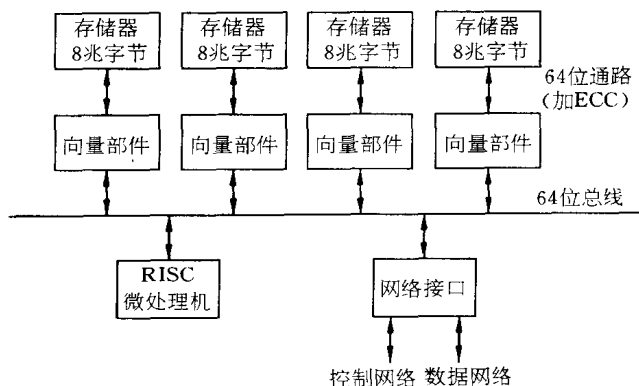


图 9.18 CM-5 中带向量部件的处理结点

峰值频宽为 128 兆字节/秒。

向量部件执行由标量微处理机发出的向量指令,并且完成所有的存储器控制功能,包括生成和检验 ECC(错误校正码)位。和传统向量处理器相似,每个向量部件有一个向量指令译码器,一个流水 ALU 和 64 个 64 位的寄存器。

每条向量指令可能会传送给一个指定的向量部件、或一对向量部件、或同时广播给所有 4 个向量部件。标量微处理机负责地址转换和循环控制,与向量部件的操作并行执行。向量部件的存储器总频宽为 512 兆字节/秒,每个结点的峰值性能为 128Mflops。在这个意义上说,CM-5 的每个处理结点本身就是一台超级计算机。16K 个处理结点的总峰值性能为 $214 \times 27 = 221\text{Mflops} = 2\text{Tflops}$ 。

控制处理机和处理结点原来都采用 SPARC 微处理器。随着处理器技术的发展,将来可能会使用新的处理器。网络的系统结构设计时,除网络接口外,要做到与选择的处理器无关。但当使用新的处理器时,网络可能还需要作少量的修改。

9.4.2 SMP

1. 概述

SMP 称为共享存储型多处理机(shared memory multiprocessors),也称为对称型多处理机(symmetry multiprocessors)。

共享存储型多处理机有三种模型:均匀存储器存取(uniform-memory-access,简称 UMA)模型、非均匀存储器存取(nonuniform-memory-access,简称 NUMA)模型和只用高速缓存的存储器结构(cache-only memory architecture,简称 COMA)模型,这些模型的区别在于存储器和外围资源如何共享或分布。

UMA 多处理机模型如图 9.19 所示。图中,物理存储器被所有处理机均匀共享。所有处理机对所有存储字具有相同的存取时间,这就是为什么称它为均匀存储器存取的原因。每台处理机可以有私有高速缓存,外围设备也以一定形式共享。

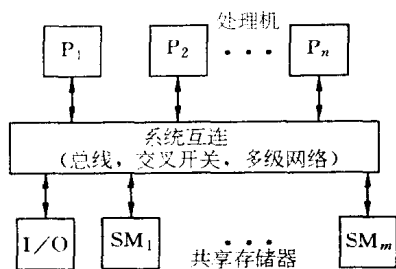


图 9.19 UMA 多处理机模型

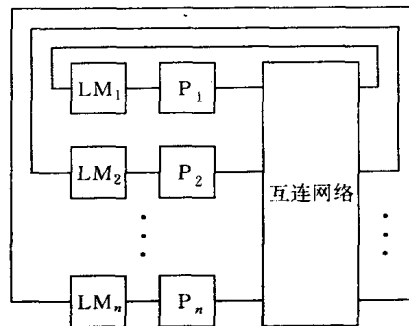


图 9.20 NUMA 多处理机模型

NUMA 多处理机模型如图 9.20 所示,其访问时间随存储字的位置不同而变化。其共享存储器物理上是分布在所有处理机的本地存储器上。所有本地存储器的集合组成了全局地址空间,可被所有的处理机访问。处理机访问本地存储器是比较快的,但访问属于另

一台处理机的远程存储器则比较慢,因为通过互连网络会产生附加时延。

COMA 模型如图 9.21 所示,它是一种只用高速缓存的多处理机。COMA 模型是 NUMA 机的一种特例,只是将后者中分布主存储器换成了高速缓存,在每个处理机结点上没有存储器层次结构,全部高速缓冲存储器组成了全局地址空间。远程高速缓存访问则借助于分布高速缓存目录进行。

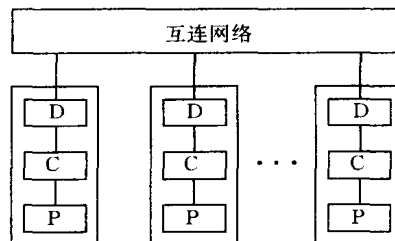


图 9.21 COMA 多处理机模型
(P:处理机 C:高速缓存 D:目录)

共享存储系统拥有统一寻址空间,程序员不必参与数据分布和传输。早期的并行处理系统几乎都是基于总线的共享存储系统,它们的发展得益于两方面的原因:一个是微处理器令人难以置信的性能价格比,另一个是在基于微处理器的并行处理系统中广泛使用的 Cache 技术。这些因素

使得将多个处理器放到同一条总线上,共享单一存储器成为可能,并通过 Cache 将所有处理器访问存储器所需的存储带宽降低到可以接受的水平。Cache 一致性是通过监听协议实现的。这种实现方式虽然简单,但是阻碍了系统的扩展能力。

到 80 年代中期,对可扩展的多处理器系统的需求不断增长。基于总线的、Cache 一致性、共享单一存储器的机器显然是不可扩展的。1996 年随着 SGI Origin 2000 系列服务器的推出,一种称为 S2MP 的并行计算机体系结构受到了广泛的注意。S2MP 全称为可扩展共享存储多处理(scalable shared-memory multiprocessing)技术。S2MP 系统将大量高性能微处理器连接起来,共享一个统一的地址空间,较好地解决其它并行处理系统无法解决的问题。

S2MP 是一种共享存储的体系结构,和大规模的消息传递系统相比,它支持简单的编程模型,系统使用方便,是对 SMP 系统在支持更高扩展能力方面的发展。共享存储系统降低了通信的额外开销,因此系统也可以运行细粒度的应用。S2MP 着眼于扩展性能的研究,和传统的基于总线的共享存储并行处理系统相比,它不存在系统中可以连接的处理器数目的总线带宽的限制。

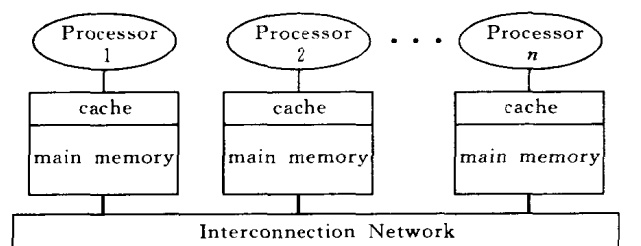
S2MP 作为大规模多处理系统,主要问题仍然是解决系统的可扩展和易编程能力。S2MP 系统采用分布式存储器技术,引入 Cache,降低了访存时延。同时系统内处理器通过高速无阻塞的互连网络连接,增加了系统的通信带宽。这些技术保证了 S2MP 系统的可扩展性能。另一方面,解决编程问题的主要方法仍然是提供与单机类似的统一地址空间,因此 S2MP 系统采用共享存储的存储器模型,每个处理器结点都可以直接访问所有的存储单元,程序员不用在程序中显式地控制在处理器之间分布数据和进行通信,因而容易编程。直接访存也使得在处理器间动态分配任务,实现负载均衡简单了。同时,由于共享存储系统是由小规模并行多处理器系统演变而来的,它们具有相同的编程模型,所以那些已有的并行应用问题可以很容易地移植过来运行,解决了并行处理系统的应用程序开发问题。

2. S2MP 体系结构

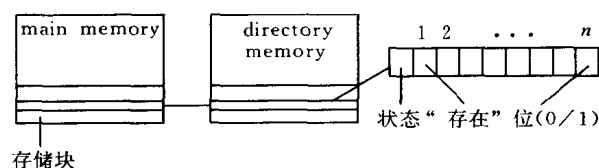
S2MP 代表了新一代提供共享编程模型的并行多处理器系统,它解决了当前小规模

共享存储多处理器系统在可扩展性上的瓶颈问题。

共享存储的多处理器系统更加接近于传统的编程模式,有效地解决了在并行处理系统上开发并行应用程序困难的问题。SMP 是这类系统的典型实现方式,但由于系统采用的互连方式限制了通信带宽和处理器数目的增加,因此系统不可扩展或扩展能力很低。S2MP 系统则从性能和扩展能力两方面解决通常基于共享存储的多处理器系统共同存在的问题,引进了复杂的存储子系统,通过硬件 Cache 对系统的共享和私有数据都进行缓存,以达到高性能。而它的扩展能力则通过两个技术手段得到保证。首先是采用了一个高速互连网络,而不是总线,提供随系统规模变化而增加的理想带宽;其次,在这些系统中采用的 Cache 一致性协议不再依赖于 snoopy 这种广播维护一致性消息的协议,基于目录的 Cache 一致性协议通过维护指向每个存储块的 Cache 缓存的指针实现一致性,这样维护一致性的操作都是对拥有某些存储块的 Cache 缓存的处理器进行的,不会占用系统太多的带宽。图 9.22 是 S2MP 系统的体系结构和 Cache 示意。



(a) S2MP 系统示意



(b) 基于目录的 cache 一致性

图 9.22 S2MP 体系结构示意图

S2MP 系统实际上是一种 NUMA 结构,每个结点由处理器和存储器两部分组成。存储器靠近处理器,而不是集中在某个地方。处理器可以访问本地存储器获取数据。NUMA 结构可以降低平均访存时延,并且随处理器数目的增加自动增加存储器带宽,也就是说存储带宽是可扩展的。由于在某个结点上访问本地存储器可以和其它结点板上的访存并行进行,系统的总带宽可以随着系统的规模而扩充。

对存储器进行 Cache 是获得高性能的一个重要因素。在最近的 10 年内,微处理器的周期从 100~200ns 降低到了少于 4ns,大约降低了 25 倍。与此同时,DRAM 访问时间只是从 150~180ns 改善至 60~80ns,大约为 3 倍。这个差距只能通过包含不同层次的 Cache 的多层次存储器模型来弥补,获得比较合理的平均访存时间。Cache 在多处理器系统对访存的速度改善更加明显,但也更加复杂,因为它们引进了一致性问题。为了实现可扩展性,目前主要的 S2MP 系统都在每个处理器结点上增加一个目录存储器,通过目录存储器保证 Cache 一致性。目录存储器的每个记录对应主存的一个存储块。状态位对

应存储块的缓存状态,可以是 uncached/shared/dirty 等。每个“存在”位对应一个结点的 Cache,标志该 Cache 是否有本记录对应存储块的缓存。目录的适用不再需要广播访存消息,为维护一致性付出的开销只出现在那些拥有数据缓存处理器节点上。S2MP 系统同时运行页迁移算法,把某个处理器经常使用的数据移动到该节点的本地存储器上,缩短了发出请求的处理器和被请求的数据的距离。这样做的好处是系统的总时延降低了,而存储带宽增加了。

3. SGI Origin2000 系列服务器

SGI 公司将 Cray Research 子公司的开关网络技术应用到 S2MP 系统中,将 SMP、MPP 以及工作站机群系统的优点结合起来,推出 Origin 2000 系列可扩展服务器产品,包括 Origin 200、Origin 2000 Deskside、Origin 2000 Rack 和 Cray Origin 2000 四种机型。Origin 200 服务器是入门级的塔式系统,具有中等扩充能力,最多可以达到 4 个处理器。Origin 2000 Deskside 桌边服务器系统支持的处理器数目最多为 8 个,Origin 2000 Rack 机柜服务器系统支持的处理器数目最多为 16 个,而 Cray Origin 2000 服务器系统具有大规模的扩充能力,支持的处理器数目可以到 128 个。这一系列服务器产品具有 SMP 的易编程和平稳扩充特性、MPP 的高度可扩展性以及工作站机群系统的可用性和简易扩展性,应用非常广泛,能满足用户日益增长和经常变化的业务需求。Origin 2000 服务器综合平衡了性能、可扩展性、可用性和兼容性,成为企业的理想信息中心,可以开展 Web 服务、数据仓库、可视服务、科学计算、图象处理和仿真等多种应用,并作为大中小企业、银行、商业和政府机构的信息管理服务器。基于 S2MP 的 Origin 2000 系统可以方便地构造灵活多变的系统和开放的软件,提高客户应用系统的质量。Internet 的爆炸和企业内部与外部合作的日益增加正在改变人们交换信息的方式。Origin 2000 服务器系列的 I/O 带宽可达 102GB/s,系统传输速度比同类 SMP 服务器快几十倍,从而可以很理想地产生、存储和传输各种数字多媒体信息。

Origin 2000 服务器系统是 S2MP 体系结构的典型实现,系统的地址空间成指数增长,并且是无缝可扩展的,从而可以根据需要决定系统规模。它的关键技术包括 CrayLink 和 Cellular IRIX 操作系统。CrayLink 消除了 SMP 技术的主要瓶颈,替代总线作为处理器结点底层互连网络。Origin 2000 系统的 S2MP 体系结构最多可以扩展至 1024 个处理器,与其它 SMP 体系结构相比,S2MP 明显具有多处理器系统所要求的高带宽和低时延。采用 S2MP,系统带宽可以线性增加,而访存时延却是非常低的。CrayLink 是一种多重交叉开关互连技术,用于连接处理器、存储器、I/O 设备等。CrayLink 使 Origin 2000 系统成为模块化系统,系统规模可以是一个基本的模块,也可以是一定数目模块的互连,有效地保护用户的投资。通过 CrayLink,分布在每个处理器结点上的存储器在逻辑上形成单一寻址空间的共享存储器系统。Cache 一致性维护因此采用基于目录的协议,在每个结点上增加了目录存储器。以 S2MP 为基础的 Origin 2000 系统在各个方面都可呈线性变化,包括计算能力、内存容量和带宽、系统互连带宽、I/O 带宽和网络连接能力。例如,Origin 2000 支持几十个 XIO 接口,这些接口都符合 SGI 高速 I/O 规范;而每种接口都支持 1.25GB/s 的带宽。Origin 2000 服务器能够把硬件和软件的故障分离,提供不同寻常的有效性、灵活性和总体性能。

Cellular IRIX 是工业界最早投入使用的蜂窝式操作系统,将操作系统功能分布到各个处理器结点上,可以实现从小系统到大系统的无缝扩展。Cellular IRIX 是从 SGI 的 IRIX 演变过来的,是以 UNIX 为基础的 64 位蜂窝式操作系统。Cellular IRIX 把多个相同的操作系统核心功能分别放到多个“蜂窝”(也就是一个操作系统单元)中,每个蜂窝管理服务器多处理器的一个子集。每个操作系统单元都可以非常有效地扩展,单元之间互相通信,从而为用户提供一种单一的操作系统接口。操作系统的这种蜂窝结构与积木式的 S2MP 硬件结合起来,提供了故障隔离能力,并使故障局限于个别操作系统单元中,提高了服务器的可用性和可靠性。Cellular IRIX 操作系统的第一版 IRIX 6.4 用于 Origin 2000 服务器,最低支持 32 个处理器,可以扩展到 128 个处理器。IRIX 6.4 采用新的算法进行数据管理、调度和输入/输出。在内存管理方面使用虚拟内存机制降低内存对象的同步粒度以适用于相对小的和本地对象,没有存储量很大、全局的、高度竞争的页面 Cache,而是每个内存对象具有一个小的页面 Cache,所有大的全局数据结构都转换成带有本地同步机制的各结点上的数据结构。在内存分配方面,IRIX 6.4 充分考虑了底层结构的拓扑特征来管理系统资源和应用程序,减少共享数据时可能出现的时延。

下面详细介绍 Origin 2000 系列服务器的结构。

(1) 结点板

结点板(也就是 Origin 200 的主板)是 Origin 2000 系统的基本构成模块,一个结点板就是 Origin 2000 服务器的一个处理器结点,其结构如图 9.23 所示。每个结点板有一到两个 R10000 处理器、一个二级 Cache、主存、用于 Cache 一致性维护的目录存储器以及用于互连的称为 HUB 的 ASIC 芯片、一个 I/O 接口、一个到互连网络的路由器接口。

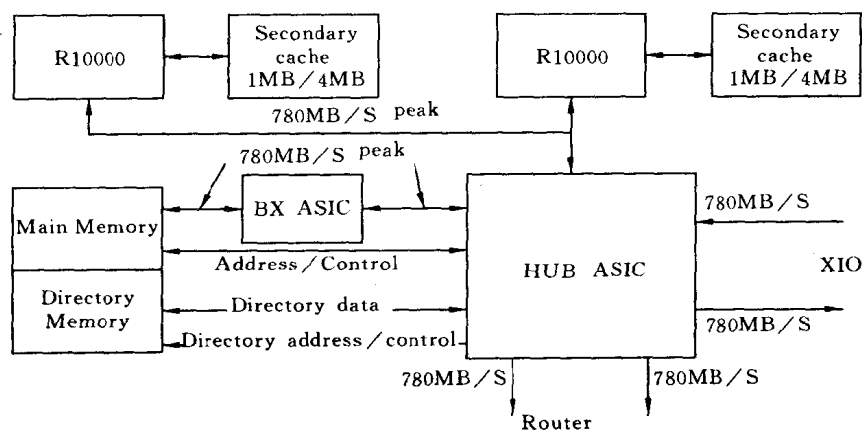


图 9.23 Origin 2000 结点板结构

结点板上的 R10000 处理器是 RISC 处理器,带 1MB 或 4MB 的二级 Cache。带 1MB Cache 的 R10000 处理器主频为 180MHz,带 4MB Cache 的 R10000 处理器主频为 195MHz。Origin 2000 系统是 CC-NUMA 存储结构,系统的主存分布在各个结点板上,但可以为所有的处理器所共享。对某个结点板上的处理器而言,在该结点板上的存储器称为本地存储器,在其它结点板上的存储器称为远程存储器。对任何一个处理器来说,系统的

主存是由很多存储块构成的单一寻址的存储空间。

结点板上的 HUB 用来连接处理器、主存和目录存储器以及接入系统的互连网络。其结构如图 9.24 所示。HUB 有一个专门的路由器端口和 I/O 端口。HUB 是四端口的交叉开关,有四个双向的端口,每个端口单向传输速率达 800MB/s(峰值带宽),因此双工时带宽可达到 1.6GB/s。这四个端口分别连接到处理器、主存、XIO I/O 和通过一个路由器连接至 CrayLink 互连网络。四个端口在内部以交叉开关互连,通过发送消息进行通信。HUB 控制每个结点子系统内部的通信,也控制和其它结点板的 HUB 进行通信。HUB 对内部消息的 request/reply 格式和外部消息 Crosstalk 或 CrayLink Interconnect 格式进行转换。所有的内部消息都由处理器和 I/O 设备初始化。消息可以分为请求(request)和应答(reply)两种。每个输入和输出 FIFO 在逻辑上分为两个队列,一个用于处理请求,一个用于处理应答。Cache 一致性协议有单独的逻辑请求和应答路径,因此保证可以避免死锁。这些消息(read, write 等)被相应的接口转换成 CrayLink Interconnect 请求,并送达目标结点的存储器或 I/O 接口。依靠目录当前状态,目标结点或者回送应答或者发送新的请求,如插入或失效消息到本地 HUB 和其它结点板的 HUB 上的其它接口。

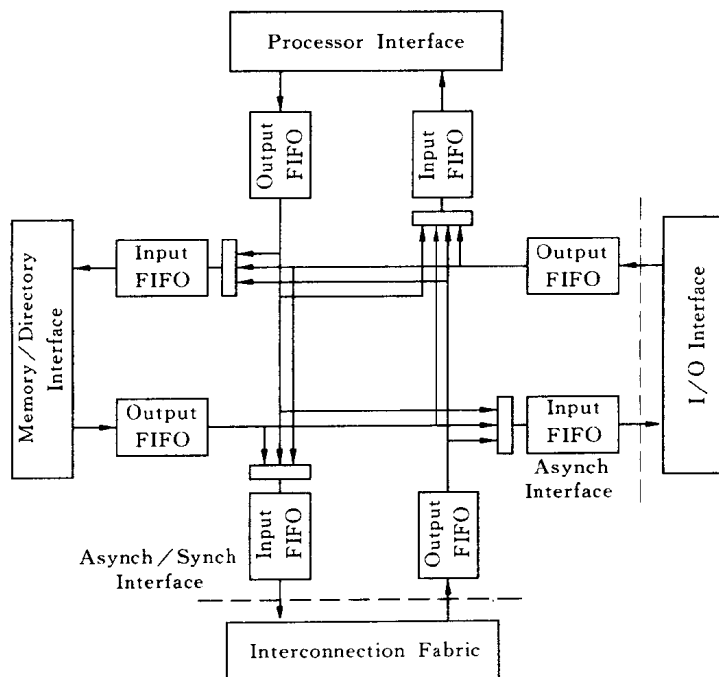


图 9.24 HUB ASIC 结构

(2) I/O 子系统

Origin 2000 系统的 I/O 子系统由一组高速的 Crosstalk (XTALK) 链路构成, Crosstalk 支持很多 SGI 和第三方的 I/O 设备。Crosstalk I/O 是分布的,在每个结点板上有一个 I/O 端口,可以被每个处理器访问。I/O 通过结点板上的单端口 Crosstalk 协议的链路控制,或者通过在 Crossbow (XBOW) ASIC 芯片上的智能交叉开关互连。XBOW

ASIC芯片将单个Crosstalk I/O 端口扩充到8个端口,6个用于I/O,2个用于连接到结点板。图 9.25 给出一个XBOW ASIC 8个端口的连接方式。

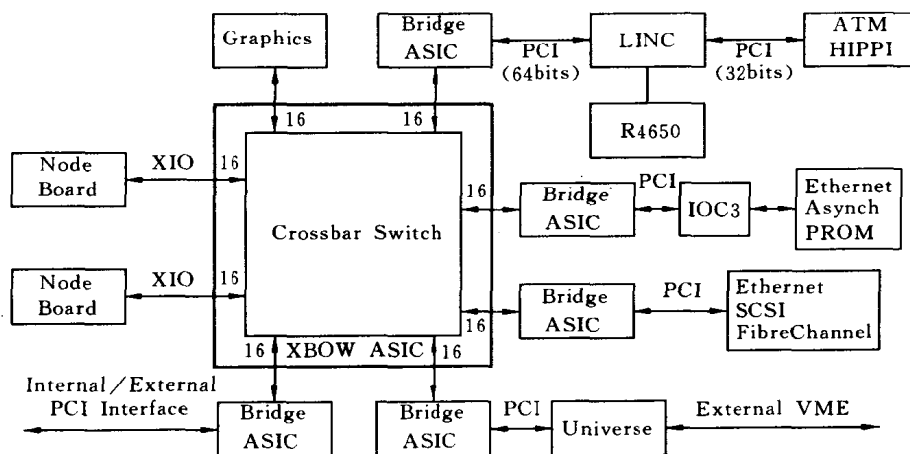


图 9.25 XBOW ASIC 8 端口连接示意

作为Crossbow的连接设备,可以是PCI、VME、SCSI、Ethernet、ATM、FibreChannel和其它I/O设备。在Crosstalk协议控制下,可以编程进行8位或16位通信。XBOW ASIC还有动态交叉开关,可以扩展两个结点板上的Crosstalk端口连接成6个16位的I/O端口,每个I/O端口可以运行在8或16位模式。在每个结点板上至少要连接一个Crossbow端口,最多是两个。具体的连接如图9.25所示。在双结点板的配置中,余下的6个XIO端口静态地分配给两个结点板,如果其中一个结点不工作了,另一个结点可以随后编程控制所有的Crosstalk端口。XBOW ASIC芯片有一个交叉开关,动态地将每个端口连接到指定的I/O设备(如结点板、图形板、串行I/O)等,如图9.25所示。Crossbow对Crosstalk消息进行解码,获得控制和目标的信息。连接到XBOW ASIC的设备称为XIO

表 9.1 Origin 2000 支持的XIO设备

XIO 设备类型	支持端口数
BaseIO	110/100Base-TX 2 460K baud serial 1 external SE Ultra SCSI 1 internal SE Ultra SCSI 1 real-time interrupt
Faster Ethernet and serial	4 10/100Base-TX 4 115kbs Serial
UltraSCSI	4 differential
FibreChannel	2 100Mbyte/sec
ATM	4 OC3 155Mbps
HIPPI-Serial	1 200Mbyte/sec

设备,Origin 2000 系统提供的 XIO 设备到 XBOW ASIC 芯片的接口板,接口板的尺寸有 $10'' \times 6.5'' \times 1''$ 和 $10'' \times 13'' \times 1''$ 两种。表 9.1 列出了 XBOW ASIC 支持的 XIO 设备。

(3) 互连网络子系统

互连网络子系统是由一组开关组成的,称为路由器,通过电缆连接成各种拓扑结构。它和通常的总线有以下的不同:互连网络是由很多的点到点的链路通过路由开关连接而成的 mesh 网,这些链路和开关允许多个传输同时发生;链路开关速度极高,每条双向链路带宽峰值达到 1.6GB/s;互连网络不需要仲裁,也不存在竞争;在增加结点板的同时,接入更多的路由器和链路,提高了互连网络的总带宽,而共享总线的带宽是固定的,不能扩展。互连网络提供了在每对 Origin 2000 系统结点之间至少两条独立的链路进行通信,这种方式使系统可以绕过不能运行的路由器和断开了的链路。每条链路进行 CRC 校验,并且按照链路层协议运行,可以重试任何失败的传输,并提供容错能力。图 9.26 所示的是连接 8 个结点板而成的 hypercube。R1 和 R0、R2 和 R3、R4 和 R6 以及 R5 和 R7 的通信可以同时进行,不会中断任何其它结点的运行。

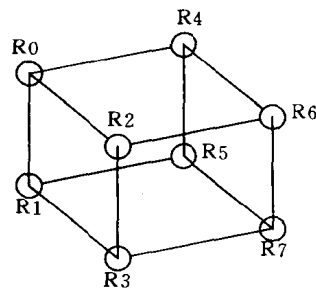


图 9.26 连接 8 个结点板的 hypercube

路由器将结点板上的 HUB ASIC 物理上连接到 CrayLink Interconnect 上。CrayLink Interconnect 为结点提供了高带宽、低时延的互连机制。路由器实现了一套路由协议,保证即使在链路出错时仍能可靠地交换数据,进行流量控制和优先级管理。路由器的核心是实现 6 路无阻塞交叉开关的路由 ASIC 芯片。路由器的交叉开关允许 6 个路由端口全双工同时操作,每个端口有 2 条单向的数据通路。图 9.27 是路由 ASIC 芯片的结构图。它的

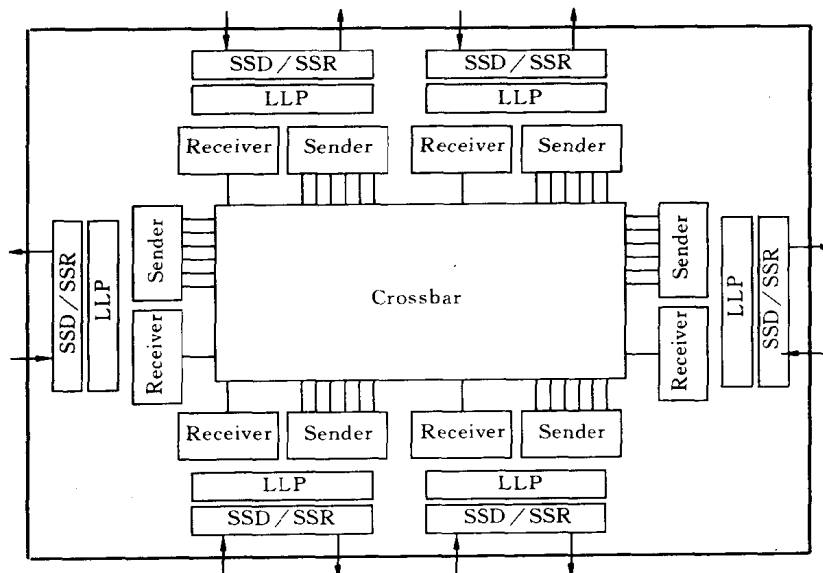


图 9.27 路由器的 Router ASIC 结构

功能有:选择发送和接收端口的最高效连接,保存接收到的数据,动态地切换 6 个端口的连接;在 CrayLink Interconnect 的链路层协议(LLP)控制下与其它路由器和 HUB 进行可靠通信;为了减少时延,消息以 wormhole 的方式通过路由器;缓存 CrayLink 信息。路由器提供的峰值通信带宽达到 9.36GB/s。

(4) 分布式共享存储地址空间

Origin 2000 服务器具有一个统一的共享地址空间,在该空间的存储器分布在所有的处理器结点中,并且可以通过互连网络被所有处理器访问。将存储器分布使得处理器访问本地存储器所花的时间比访问远程存储器时间少,系统总的访存时延降低了。尽管存储器在物理上是分布的,主存对所有的处理器都是可访问的。I/O 设备也是在一个共享的地址空间上分布的,每个 I/O 设备在整个系统中是全局可寻址的。Origin 2000 服务器的存储系统是一个复杂的子系统,共分为四个层次。最接近处理器的是寄存器,和 CPU 在同一块芯片上,访问时延时间最少;第二层是 Cache,有主 Cache 和二级 Cache 两层,主 Cache 在 R10000 芯片上,二级 Cache 在和处理器紧耦合的另一块板上;第三层是本地存储器,和处理器同在一块结点板上,包括主存和目录存储器;第四层是远程 Cache,用于存放指定的存储块的缓存。Cache 用于减少访问共享存储空间所需的时间,即访存时延,通常采用 SRAM 存储器。尽管数据只在本地或远程存储器上存放,它们的缓存可以在多个结点板上的 Cache 中,使缓存保持一致是 HUB 的任务,实现 Cache 一致性的逻辑处理过程称为 Cache 一致性协议。

(5) Cache 一致性

在 Origin 2000 服务器系统中,数据可以被所有的处理器和它们的 Cache 缓存和共享,将数据存入 Cache 可以减少访存时延,但是也使得维持数据的一致性复杂了。Cache 一致性协议就是设计用来维护数据一致性,保证不管哪个处理器上请求了该数据,得到的都是数据的最新版本。Origin 2000 服务器采用基于目录的一致性协议。每个存储块有一个相应的目录项,这些目录项存放在称为目录的表格中。由于存储器在整个系统中是分布的,所以目录也是分布的。每个目录项存放的信息是存储块的系统全局缓存状态和指向每个对该存储块进行缓存的 Cache 的位向量指针。图 9.28 给出了存储块和 Cache 的关系。通过检查状态和位向量,存储器可以确定哪些 Cache 在某次访存操作中需要参与操作以

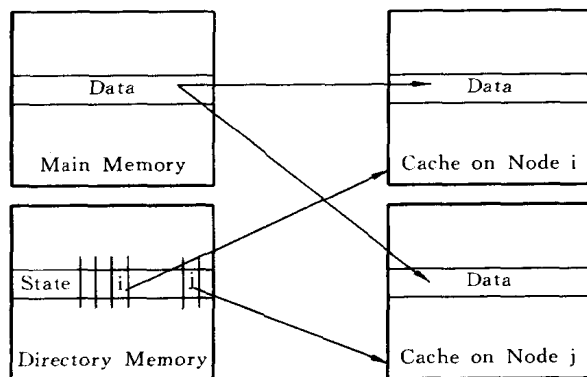


图 9.28 基于目录的 Cache 一致性

保持一致。存储块有四种状态,即 unowned(该存储块的缓存不存在)、exclusive(只有一个 Cache 缓存)、shared(有多个 Cache 缓存)、poisoned(该存储块所在页已经迁移到其它结点)。基于目录的 Cache 一致性避免了 snoopy 协议占用太多带宽的缺点,它不需要每个处理器对每次访存事件都进行广播。相反,只有那些存放有被访问存储块数据的 Cache 需要被通知参与存储访问操作,也只有这些 Cache 会受到影响。这个特点保证了系统不会因为支持 Cache 一致性而影响了带宽的扩展性能。基于目录的协议可以采用两种方法维护一致性,即更新和失效。Origin 2000 系统选择了后一种方法。

(6) 系统时延

访存时延(memory latency)是系统完成一次读或写操作所用的时间,它由两个参数构成,即访问时间(access time)和周期时间(cycle time)。访问时间是从数据请求发出到数据返回的总时间,周期时间是两次请求的间隔时间。降低访存时延的方法有:

- ① 在 Cache 中缓存经常访问的存储块;
- ② 对存储器进行分布,把常用数据移动到靠近处理器的存储层次上;
- ③ 页迁移,将经常访问的数据从远程处理器迁移到本地存储器;
- ④ 优化结点板设计和 CrayLink 的拓扑结构,减少数据需要通过的交叉开关数目和到达远程存储器的路径长度。

表 9.2 列出了在不同配置下系统存储带宽和访存时延。

表 9.2 系统存储带宽和访存时延

配置	存储器和 I/O 带宽 (峰值、GB/s)		访存时延(ns)	
	存储器	I/O	最大值	平均值
I/O : Node : CPU				
1 : 1 : 2	0.59	1.25	313	313
2 : 2 : 4	1.19	2.50	497	405
2 : 4 : 8	2.38	4.99	601	528
4 : 8 : 16	4.75	9.98	703	641
8 : 16 : 32	9.50	19.97	805	710
16 : 32 : 64	19.00	39.94	908	796
32 : 64 : 128	38.00	79.87	1112	903

(7) 系统带宽

描述系统带宽有三种情况。峰值带宽是理论值,直接将时钟频率和接口的数据宽度相乘得到;持续带宽是峰值带宽扣除数据包头部消息和其它额外开销得到的结果,它也称为峰值负载带宽,没有考虑到竞争和其它可变因素;平分带宽是指在将互连网络等分后,数据在等分网络上的传输速率,在数据没有被优化存放时可以比较准确地描述数据传输率。每个 HUB 链路的峰值带宽是 1.6GB/s(双向),每个 8 位的 HUB Crosstalk 端口的峰值带宽是 800MB/s(全双工)。每个 8 位的 HUB Crosstalk 端口的持续带宽是 355MB/s(单向),全双工时为 640MB/s,单向的比全双工的一半高是因为它没有响应另一方向来的数据包所需的时延。每个 16 位 HUB Crosstalk 端口的持续带宽是 711MB/s(单向),全双工

为 1.28GB/s。HUB 的存储器和处理器端口的峰值带宽是 800MB/s。如表 9.3 所示。表 9.4 列出了系统的平分带宽。

表 9.3 系统的峰值带宽和持续带宽

连接到 HUB 的部件	半双工/全双工	峰值带宽 (/s)	持续带宽 (/s)
处理器	单向	780MB	624MB
存储器	单向	780MB	680MB
互连网络	全双工	1.56GB	1.25GB
	半双工	780MB	693MB
XIO 板	16 位全双工	1.56GB	1.25GB
	16 位半双工	780MB	693MB

表 9.4 Origin 2000 系统的持续(峰值)平分带宽

系统规模(处理器数目)	平分带宽(GB/s)	平分带宽(GB/s)
	(没有 Express 链路)	(有 Express 链路)
8	1.25 (1.56)	2.5 (3.12)
16	2.5 (3.12)	5.0 (6.24)
32	5.0 (6.24)	10 (12.5)
64	10 (12.5)	N/A
128	20 (25)	N/A

(8) 扩展连接方式

Origin 2000 服务器系列主要有四种机型,即 Origin 200、Origin 2000 Deskside、Origin 2000 Rack 和 Cray Origin 2000,它们的最大可支持的处理器数目分别为 4、8、16、128。但是,由于 Origin 2000 系统是模块化系统,由结点板通过路由器连接可以构成满足应用要求的许多种处理器数目配置的系统。图 9.29 给出 4、16 和 32 个处理器配置时的系统互连拓扑结构。图中 P 指的是处理器,N 指的是结点板,H 表示 HUB,R 表示路由器。图 9.30 是 64 以及 65~128 个处理器的连接方式。

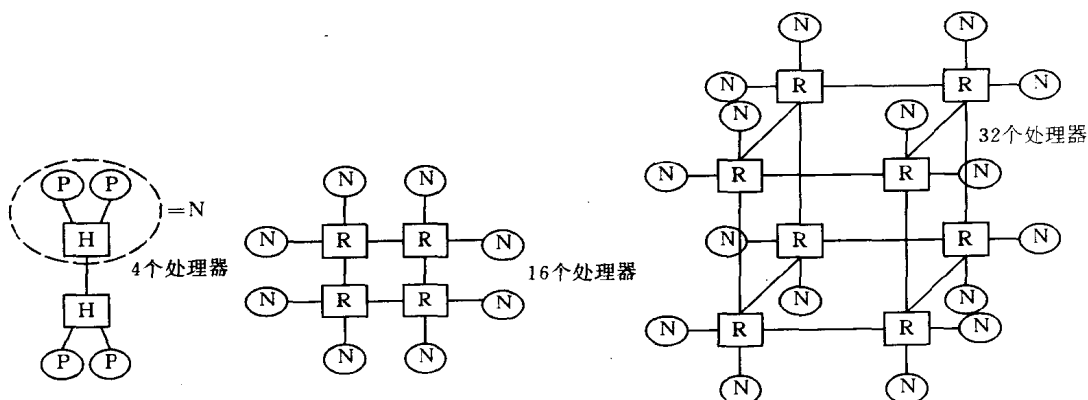


图 9.29 Origin 2000 系统连接图

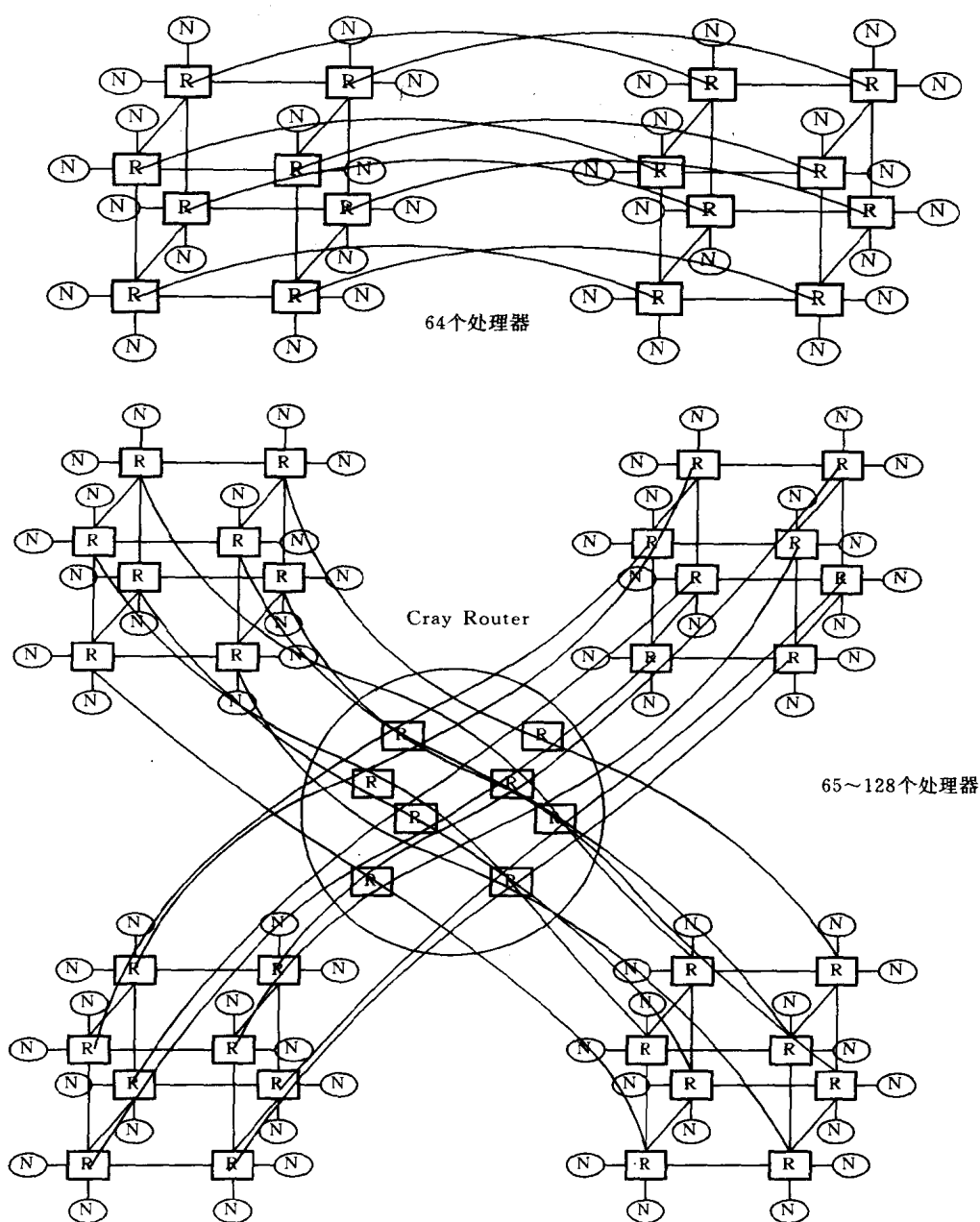


图 9.30 Origin 2000 系统连接图

可以看出,系统的扩展是相当容易的,主要得益于系统的CrayLink互连网络机制。两个结点板通过HUB直接连接得到4个处理器的机器,相当于将两台Origin 200服务器直连使用。由于路由器提供了两条连接结点板的链路,由一个路由器和两个结点板构成一个模块,在模块的基础上,充分利用路由器的其它四个接口可以扩展系统到不同的规模。使用其中的两条链路,可以得到最多16个处理器的机器配置。使用其中三条链路,形成一

个 cube, 可以达到 32 个处理器的配置。当把所有的链路都利用上, 系统的最大配置可以达到 128 个处理器, 专门增加 Cray Router 路由器将 4 个由 32 个处理器构成的 cube 连接成 hypercube 的拓扑结构。

(9) 性能评价

下面给出 Origin 2000 服务器运行一些标准测试集得到的性能指标, 所有的数值都采用 195MHz 的 R10000 处理器, 带有 4MB Cache。第一组性能指标是 SPEC95, 它反映系统整数和浮点数的计算性能; 第二组性能指标是 SPECrate95, 它计算系统同时运行 n 个相同的 SPEC95 测试例子的总时间, 反映系统单个处理器的计算容量; 第三组性能指标是 Linpack, 用来评价系统的浮点运算性能, 适用于单处理器和多处理器两种情况; 第四组是 NAS Parallel Benchmark, 可以给出并行处理系统的加速比数值。以上各组性能列于表 9.5 至 9.8 中。

表 9.5 SPEC95

SPEC95	最好情况	基本情况
SPECint95	9.43	8.57
SPECfp95	19.1	17.5

表 9.6 SPECrate95(32 个拷贝)

SPECrate95	最好情况	基本情况
SPECrate_int95	2640	2411
SPECrate_fp95	3814	3518

表 9.7 Linpack(单位: MFLOPS)

Linpack 100×100	1CPU: 113.6
Linpack 1 000×1 000	1CPU: 344
Linpack N×N(15k×15k)	32CPU: 8987

表 9.8 NAS Parallel Benchmark V.1

CPU	Class A					Class B				
	1	16		32		1	16		32	
测试集	时间 (秒)	时间 (秒)	加速比	时间 (秒)	加速比	时间 (秒)	时间 (秒)	加速比	时间 (秒)	加速比
EP	101.9	6.71	15.19	3.58	28.46	408.1	26.86	15.19	14.26	28.66
MG	30.93	3.44	8.99	2.14	14.45	192.8	14.78	13.04	8.57	22.50
CG	27.85	2.04	13.65	2.14	13.01	1515.7	159.9	9.48	80.29	18.88
FT	49.63	4.22	11.76	2.77	17.91	714.9	48.95	14.60	25.73	27.78
IS	13.94	1.34	10.40	1.12	12.45	112.3	9.22	12.18	5.3	21.19

续表

CPU	Class A					Class B				
	1	16		32		1	16		32	
测试集	时间 (秒)	时间 (秒)	加速比	时间 (秒)	加速比	时间 (秒)	时间 (秒)	加速比	时间 (秒)	加速比
LU	334.3	29.83	11.21	19.22	17.39	1512.2	132.9	11.38	89.12	16.9 7
SP	671.1	53.25	12.60	31.06	21.61	3219.2	240.3	13.40	146.8	21.9 3
BT	747.9	57.10	13.10	34.26	21.83	3477.6	253.7	13.71	155.9	22.3 1

9.4.3 机群系统

9.4.3.1 机群系统的组成、特点和关键技术

1. 机群系统的组成

机群系统是利用高速通用网络将一组高性能工作站或高档 PC 机,按某种结构连接起来,并在并程序计以及可视化人机交互集成开发环境支持下,统一调度,协调处理,实现高效并行处理的系统。从结构和结点间的通信方式来看,它属于分布存储系统,主要利用消息传递方式实现各主机之间的通信,由建立在一般操作系统之上的并行编程环境完成系统的资源管理及相互协作,同时也屏蔽工作站及网络的异构性,对程序员和用户来说,机群系统是一个整体的并行系统。机群系统中的主机和网络可以是同构的,也可以是异构的。目前已实现和正在研究中的机群系统大多采用现有商用工作站和通用 LAN 网络,这样既可以缩短开发周期又可以利用最新的微处理器技术。大多数机群系统的并行编程环境也是建立在一般的 Unix 操作系统之上,尽量利用商用系统的研究成果,减少系统的开发与维护费用。

RISC 技术、网络技术和并行编程环境的发展使得机群系统这一新的并行处理系统形式正成为当前研究的热点。

(1) 由于 RISC 技术的发展,使得微处理器的性能不断提高。高档芯片的运算能力平均每年增长 30%,而价格在不断下降,直接使用商用工作站或 PC 机作为运算结点的机群系统在结点性能上能够同处理器的发展保持同步增长。

(2) 网络技术的进步使得松散耦合系统的通信瓶颈逐步得到缓解。网络传输速度的提高,有效地提高了应用程序之间的通信带宽。快速以太网的速率为 100Mbps,ATM 局域网的带宽达到 155Mbps, 622Mbps 的产品也已经研制出来。而开关技术的发展则大幅度地降低了传输延迟,使得许多高速局域网能和 MPP 中的专用互连网络的性能相当。例如,由 MyriCom 公司生产的 Myrinet,提供 1.28Gbps 的双向链路,已经和专用网络的带宽接近;它的开关延迟每级只有 1 μ s。采用这种网络的机群系统加上新的协议控制机制,

点-点的往返延迟只有十几微秒到几十微秒。UIUC 研制的 Fast Message 平均往返延迟只有 $12\mu\text{s}$, UC Berkeley 的 Active Message 为 $36\mu\text{s}$, 与 CM-5 的专用网络相当, 比 Meiko CS-2 的延迟还要小。

(3) 并行编程环境的开发使得新编并行程序或改写串行程序更为容易。并行应用程序的开发和不同系统之间的可移植性一直是传统并行系统能否广泛应用的一个问题。由于机群系统的发展, 近年来开发出了多个并行程序开发及运行系统, 比如 PVM, MPI, Express, Linda, P4 等。这些系统的适应平台非常广, 现在流行的工作站上都可以运行, 应用程序在这些系统上的可移植性较好, 往往仅需要修改相应的数据交换语句。特别是 PVM 和 MPI, 由于其开放性, 许多大学和研究机构都有广泛的研究和应用, 在这些环境下开发了许多应用程序。

这些技术的进步使得机群系统这一并行处理的新的结构形式受到广泛的关注, 国外许多大学和计算机公司都在进行这方面的研究和开发工作。

2. 机群系统的特点

机群系统之所以能够从技术可能发展到实际应用主要是它与传统的并行处理系统相比有以下几个明显的特点:

(1) 系统开发周期短。由于机群系统大多采用商用工作站和通用 LAN 网络, 使结点主机及系统管理相对容易, 可靠性高。开发的重点在通信和并行编程环境上, 既不用重新研制计算结点, 又不用重新设计操作系统和编译系统, 这就节省了大量的研制时间。

(2) 用户投资风险小。用户在购置传统巨型机或 MPP 系统时很不放心, 担心使用效率不高, 系统性能发挥不好, 从而浪费大量资金。而机群系统不仅是一个并行处理系统, 它的每个结点同时也是一台独立的工作站, 即使整个系统对某些应用问题并行效率不高, 它的结点仍然可以作为单个工作站使用。

(3) 系统价格低。由于生产批量小, 传统巨型机或 MPP 的价格都比较昂贵, 往往要几百万到上千万美元。工作站或高档 PC 机由于它们是批量生产出来的, 因而售价较低。由近十台或几十台工作站组成的机群系统可以满足多数应用的要求, 而价格却比较低。

(4) 节约系统资源。由于机群系统的结构比较灵活, 可以将不同体系结构、不同性能的工作站连在一起, 这样就可以充分利用现有设备。单从使用效率上看, 机群系统的资源利用率也比单机系统要高得多。UC Berkeley 计算机系 100 多台工作站的使用情况调查表明, 一般单机系统的使用率不到 10%, 而机群系统中的资源利用率可达 80% 左右。另一方面, 即使用户设备更新, 原有的一些性能较低或型号较旧的机器在机群系统中仍可发挥作用。

(5) 系统扩展性好。从规模上说, 机群系统大多使用通用网络, 系统扩展容易; 从性能上说, 对大多数中、粗粒度的并行应用都有较高的效率。清华大学计算机系研制的可扩展机群系统上测试的结果表明 8 台工作站的加速比可以达到 $5.83 \sim 7.9$, 并行处理的效率为 $72.88\% \sim 99\%$ 。

(6) 用户编程方便。机群系统中, 程序的并行化只是在原有的 C、C++ 或 Fortran 串行程序中, 插入相应的通信原语。用户使用的仍然是熟悉的编程环境, 不用适应新的环境, 这样就可以继承原有软件财富, 对串行程序做并不很多的修改。

3. 机群系统的关键技术

对于并行处理系统,人们希望要有较高的结点运算速度,系统的加速比性能接近线性增长,并行应用程序的开发要高效、方便。目前,机群系统大多采用商用高性能工作站或高档 PC,结点的运算速度问题不是很突出,因而主要的研究方面是在提高系统的并行效率、使系统的使用更为方便上,包括建立高效的通信系统,有效地管理全局资源和提供友好的并行应用程序开发环境等。

(1) 高效的通信系统

机群系统一般使用通用局域网连接。目前常用的局域网技术大体可以分成两类,一类是共享介质网络,最常见是 10Mbps 或 100Mbps 的 Ethernet;另一类是开关网络,比如 155Mbps/622Mbps 的 ATM、640Mbps/1.28Gbps 的 Myrinet 和 100Mbps 的交换式 Ethernet。对于共享介质网络,由于其聚合网络频带与单独链路频带是一样的,其性能会随网络负载的增加而下降,特别是对于某些负载比较集中的应用程序,这种影响会更明显,但是售价便宜,组成系统也相对容易,是组成中低档机群系统的一种较好的选择。而开关网络则相反,其聚合网络频带比单独的链路频带要高得多,理论上讲是 N 倍;除开关的交换延迟影响外,性能不会随网络负载的增加而降低很多,开关网络的另一个优点是其可扩展性较好,由于 Wormhole、Cut-through 等交换技术的发展,交换延迟已经很低,与发送/接收端的开销相比要小得多。比如,Myrinet 开关的一次交换延迟小于 $1\mu\text{s}$,一个中等规模的机群系统(16~32 台)的点-点的往返延迟仅有几十微秒。但是交换开关及相应接口卡的售价要高得多,组成机群系统的价格相对也比较高,对系统的普及会有一定影响,参见表 9.9。

表 9.9 几种常用局域网的性能/价格情况

类型	速度	TCP/IP 往返延迟 (μs)	集线器/交换机 (千元)	接口价格 (千元)	实现灵活性
Ethernet	10Mbps	1 438	4/-	1	差
Fast Ethernet	100Mbps	1 347	15/85	1.5	差
ATM	155/622Mbps	1 285	-/210	15	一般
Myrinet	640/1 280Mbps	1 506	-/20	12	好

在不考虑网络负载的情况下,一般使用点-点的应用程序可见带宽和往返延迟来衡量通信系统的性能。应用程序可见带宽说明了网络的长消息包的传输性能,虽然由于网络技术的飞速发展,网络的物理链路越来越快,但是应用程序的可见带宽比起链路速度来要小得多,主要原因有网卡接口的硬件限制,协议处理开销和操作系统开销。例如,Myrinet 的物理链路是双向的 640Mbps,而在 TCP/IP 协议上点-点的应用程序可见带宽只有 38Mbps。往返延迟是 1 字节或 0 字节数据消息包的往返传输时间,它说明了网络短消息包的传输性能。新的网络技术大幅度地提高了传输速度,但往返延迟没有太大变化。从表 9.9 可以看出,快速以太网、ATM 和 Myrinet 在 TCP/IP 上的往返延迟与 10Mbps 的延迟相差不多。目前,通信系统的研究方向主要在减小往返延迟和提高链路带宽的利用率

上,实现方法有精简协议处理,开发新的通信机制和减少系统开销。

(2) 并行程序设计环境

随着 MPP 和机群系统等分布存储结构并行系统的发展,开发出了 PVM、MPI、Express、P4 等基于 Message Passing 方式的并行程序设计环境,它为并行程序的设计和运行提供一个整体系统和各种辅助工具。功能包括提供统一的虚拟机、定义和描述通信原语、管理系统资源、提供可移植的用户编程接口和多种编程语言的支持。目前研制的机群系统大多支持 PVM 和 MPI,除了能适应广泛的硬件平台和编程方便等特点之外,它们都是免费软件,可以方便地进行再开发,有利于系统的推广与应用。正是由于它们都是免费软件,所以在支持语言、容错及工具等方面都不完善,许多研究机构和大学正在做这方面的研究。

开发并行应用程序要比串行程序困难得多,它要涉及多个处理器之间的数据交换与同步,要解决数据划分、任务分配、程序调试和性能评测等问题,需要相应支持工具,比如并行调试器、性能评测工具、并行化辅助工具,它们对程序的开发效率与运行效率都有重要作用。目前,提供工具较完善的系统有 FAUST、Express、TOPSYS 和 VIDE。

(3) 多种并行语言的支持

并行程序设计语言是并行系统应用的基础,已有的机群系统大多支持 Fortran、C 和 C++,实现的方法主要是使用原有顺序编译器链接并行函数库,比如 PVM、MPI,或者加入预编译,比如 Multi-thread C, MPC++。目前机群系统并行程序设计语言的研究主要在三个方面:1) 扩展原有顺序语言,提供广泛的并行语言支持,例如,清华大学可扩展机群系统的 ADA、MPC++。2) 提供全新的并行语言,比如 Occam。3) 研究自动化并行编译方法,直接将顺序程序编译成并行代码。目前比较成功的有 UIUC 的 Polaris, Stanford 的 SUIF 和复旦大学的 AFT。

(4) 全局资源的管理与利用

有效地管理系统中的所有资源是机群系统的一个重要方面,常用的并行编程环境 PVM、MPI 等对这方面的支持都比较弱,仅提供统一的虚拟机。主要原因是结点的操作系统是单机系统,不提供全局服务支持,同时也缺少有效的全局共享方法。UC Berkeley 的 NOW 项目中提出,在一般操作系统(Unix, Linux, Windows NT 等)之上建立一个全局 Unix: GLUnix 来解决机群系统中的所有资源管理,包括组调度、资源分配和并行文件系统。一般认为其中并行文件系统对提高系统的性能潜力最大,即所谓 Terabytes >> Teraflops,就是说目前限制并行程序性能的因素主要来自 I/O 瓶颈,提高 I/O 性能的方法较之提高 CPU 速度更能增强并行系统的性能。

由于网络技术的发展,通信延迟越来越小,网络访问比本地磁盘访问要快得多。在 155Mbps/s 的 ATM 网络上,读取其它结点的内存 100MBytes 的时间是读取本地磁盘的五分之一。现在的工作站和高档 PC 机都配有相当多的内存(32Mbytes~128MBytes),整个机群系统的全部内存是一个很大的资源,利用其它结点的空闲内存作为本地结点的虚拟内存和文件缓存,可以节省相当多的访盘时间。据 UC Berkeley 的实验统计,对需要经常访盘的应用程序,使用这种方式可以比使用本地磁盘快 5~10 倍。

除了这几个主要方面的研究之外,还有许多特定应用方面的研究,比如,广播、多播等

全局操作的高效实现、DSM 并行模型的支持、并行 I/O 的研究等。

9.4.3.2 机群系统通信技术

通信子系统是并行计算机系统的重要组成部分,它完成系统中各结点之间数据传递的功能,其性能的好坏直接影响到并行计算的加速比和效率。这是因为并行计算时间是由各结点计算时间和结点间数据通信时间两部分组成,如果通信时间所占的比例过大,则必然使得并行计算的加速比下降,整个系统的效率也不会高。

并行机群系统是基于高性能工作站或高档微机和局域网(LAN)而发展起来的新系统,它是一个由若干微机或工作站通过普通 LAN 互联而成的松耦合计算机系统,这与大规模并行计算机(MPP)有较大差别,MPP 则通常是采用专用网络以紧耦合方式进行结点间的互联。与 MPP 相比,机群系统具有可扩展性好、性能/价格比高的特点,但网络带宽通常较低,如使用广泛的传统以太网的带宽只有 10Mb/s,即使是新出现的一些高速网络的带宽也只不过上百 Mb/s,另外,局域网 LAN 所使用的通信协议通常是 TCP/IP 协议,这种协议处理开销比较大,影响了网络硬件特性的发挥;而 MPP 的内部网络带宽通常都在数百 Mb/s 以上,而且是采用专用的通信协议,如 Paragon XP/S 的网络带宽可达到 1.4Gb/s,其通信协议是 NX 通信库。可见,机群系统中性能过低的通信系统会影响到整个并行计算效率的提高,因此要大力发展并行机群系统,一个关键的问题是对其通信技术进行深入的研究,以期大幅度地提高网络通信系统的性能。

1. 影响通信系统性能的因素

机群系统下的通信子系统的性能是整个系统的薄弱环节,在介绍提高通信系统性能的方法措施之前,先从网络硬件、通信软件两方面分析影响通信系统性能的主要因素。

(1) 网络带宽低

机群系统使用的网络是普通的局域网,而局域网的带宽通常都比较低,如传统以太网的带宽只有 10Mb/s。局域网的带宽之所以低,原因主要是局域网是为长距离的数据通信而设计的,由于通信距离较长,限制了通信速度的提高,因为信号的频率越高,它能够传输的距离也越短。另外一个原因是出于价格上的考虑。为了降低网络系统布线所需的成本,大多数 LAN 共享一根信号总线进行数据传输,因此这也在很大程度上影响了网络系统的性能,特别是在网络负载较重时,由于各结点都要抢占信号总线,很容易造成通信阻塞,使得实际通信带宽比其最大带宽要小得多。

(2) 传统 TCP/IP 协议的多层次结构带来了很大的处理开销

TCP/IP 协议是面向低速率、高差错和大数据包传输而设计的,它是一个多层次的软件结构,按自底向上的顺序划分,可分为四层:网络接口层、网间网层(IP)、传输层(TCP)和应用层。由于协议层次多,在进行数据传输时,数据需要经过多次拷贝才能从应用层传递到网络接口或从网络接口传送到应用层,而多次的拷贝带来了很大的网络延迟时间。另外,在多层协议的实现中,各层还重复实现了很多相同的功能,比如:

- 从 IP 层到传输层都要进行差错控制
- 从网络接口层到应用层都要进行协议的处理机调度
- 从 IP 层到应用层都要进行流量控制

- 从 IP 层到应用层都要进行数据包组装和定序的缓冲

这些冗余的功能虽然可确保数据的无差错传送,但随着链路传输出错率从 10^{-4} 降至 10^{-9} ,这种冗余处理反而限制了数据及时提交给应用程序处理。可见,多层次的协议结构是造成通信瓶颈的主要原因之一,合并某些层次,删除冗余的处理,设计一种轻型通信协议,是提高通信性能的重要方法。

(3) 协议复杂的缓冲管理增加了网络延迟

网络协议处理包括很多功能,如流量控制、差错控制、出错重发机制、拥塞控制等,而这些功能的实现都与缓冲管理密切相关。

缓冲管理的作用是完成数据的分组和组装,缓冲区可看成一种网络资源,这种资源是有限的,对它的管理很重要。不过通常的缓冲管理机制都比较复杂,例如,Berkeley UNIX 采用一种叫 mbufs 的结构对协议的数据包进行缓冲管理,但 mbufs 的算法很复杂,开销很大。在 DECstation 5000 上,对单字节 TCP 消息缓冲管理,mbufs 需要 100 微秒,而对 512 字节数据包需要 300 微秒,可见缓冲管理带来的网络延迟也很大,如何简化协议复杂的缓冲管理也是通信技术研究的主要内容。

(4) 操作系统额外开销不可忽视

操作系统提供的系统调用和原语是网络协议实现的底层软件支持。在网络协议实现中涉及到上下文切换、调入/调出页面、启动 I/O 设备、中断响应等操作系统处理,有时这些开销可能比协议本身的处理开销还大。比如,在 Sun 3/60 系统上 TCP/IP 对一个数据包的协议处理时间为 100 微秒,而操作系统的额外开销却高达 240 微秒,这就造成了通信性能对操作系统一定程度上的依赖。因此,要提高通信系统的性能,降低网络延迟,应当尽量减少网络协议对主机操作系统的服务请求,最大限度地使通信与计算重叠。

2. 提高通信系统性能的方法措施

(1) 采用新型高速网络,提高网络带宽

为了提高机群系统的网络带宽,必须采用新型的高速网络来取代 10Mb/s 以太网。由于多媒体应用、实时网络系统、大规模并行计算等应用对高速网络的需求,推动了网络技术的飞速发展,目前出现了多种新型的高速网络,如快速以太网、ATM、Myrinet。这些高速网络从所采用的技术上分,可分成两类:共享介质类型和基于开关类型。共享介质类型的高速网络主要有快速以太网和 FDDI;基于开关的高速网络主要有 ATM 和 Myrinet。下面对其中三种网络的性能分别进行介绍。

• 快速以太网

快速以太网是在 10Mb/s 普通以太网的基础上发展起来的一种高速网络,其通信带宽是 100Mb/s,比原来的以太网在速度上提高了 10 倍,但由于它原封不动地采用现有的载波侦听技术(CSMA/CD),使得它具有两个主要缺点:一是所有的设备共享 100Mb/s 带宽;二是实际带宽小于 100Mb/s,特别是当系统负载较重的情况下,网络性能下降的幅度比较大。不过随着开关网络技术的发展,产生了基于开关网络的快速以太网,能够较大幅度提高整个系统的聚合带宽。另外,快速以太网技术发展得很快,现在有多家公司正在研制 1Gb/s 以上的快速以太网,预计明年年底将推出商业产品,而且这种产品将使用一种新的介质访问技术来取代现在的 CSMA/CD 方式,因此它的性能有可能取得较大的

提高。

• ATM

ATM 是以 53 字节长的信元(cell)为单位进行传输的新型通信网络。它在端与端之间采用“虚电路连接”机制,而在网络中则采用高速分组交换技术。从理论上讲,ATM 不受速度限制,可以在任何速率下操作,具有较好的实时性和灵活性,尤其适用于视频、声音等多媒体数据的实时传输。但由于信元报头开销较大,降低了 ATM 的通信带宽。另外,ATM 的机制还不完善,缺乏流量控制,容易出现阻塞,可能会使实际吞吐率变得更低。目前 ATM 主要有三类标准产品:25Mb/s、100 Mb/s 和 155Mb/s 的网卡和相应的交换器模块。622Mb/s 产品也即将由某些厂家如 Bay、Cisco 推出。

• Myrinet

Myrinet 是由 Myricom 公司研制的一种高速网络。由于它采用了多项在 MPP 中使用的技术,如交换技术、Cut-through 路由技术、8 位并行数据通道等,使得它具有极高的通信带宽。其物理链路提供了一对 640Mb/s 的数据通道,据最新的资料得知,它的物理链路速度又提高了一倍,单向达到了 1.28Gb/s,双向总带宽则为 2.56Gb/s,这样高的带宽是普通以太网、快速以太网、ATM 等无法比拟的。虽然这只是最底层的物理通信带宽,即使考虑到上层软件的开销,其性能也具有相当的吸引力。Myricom 提供的数据说明,在 SPARC-2,20MHz SBus 上,Myrinet 的 API(8KB 大小的数据包)能达到 250Mb/s 的带宽。

综合以上的性能分析可看出,这些新型网络的传输速度是传统以太网的 10 倍或更高。由于高速网络的运用,使得影响通信系统性能的瓶颈已从过去的网络硬件转移到网络通信软件上,因为虽然高速网络降低了网络的传输延迟,但并没有减少通信协议的处理开销。由于通信协议处理开销过大,在很大程度上阻碍了高速网实际性能的提高。我们通过对 Myrinet 的研究就可说明这一点。Myrinet 的物理链路带宽为 640Mb/s,而在 TCP/IP 协议层测到的带宽仅有 42Mb/s。可见,上层通信协议的开销使得高速网络的高性能得不到充分的发挥。

(2) 设计新的通信协议,降低通信延迟

高速网络性能的提高,往往是指其峰值带宽。与传统以太网比,高速网的峰值带宽有了很大程度的提高,从 10 Mb/s 增长到 100 Mb/s、640 Mb/s,甚至 1Gb/s。但是峰值带宽都是在大数据包传输时得到的,对小程序包来说,其性能又如何?通过对实际系统的测试,发现结果不尽人意。表 9.10 是对 10Mb/s 以太网和 640 Mb/s Myrinet 一个字节数据包往返延迟时间的测试结果(在 Sparc 20 工作站上)。

表 9.10 以太网与 Myrinet 网络延迟时间

网络类型	TCP 往返延迟(μ s)	UDP 往返延迟(μ s)
以太网	1 438	1 146
Myrinet	1 506	1 370

Myrinet 的通信延迟反而比以太网还大。可见,高速网络在传输小程序包时的性能并

不理想,而小数据包下网络系统性能的好坏,对并行计算有很大影响,因为用户设计的并行算法经常需要进行小数据量的数据交换,而且并行计算环境,如 PVM,在运行时也需发送很多数据量很小的控制消息进行系统运行情况的监护。对其它方面的应用而言,小数据包的传输延迟也起到举足轻重的作用。例如,UC Berkeley 通过对 NFS 数据包的跟踪发现,95%的数据包的大小不超过 192 字节,数据包的平均大小是 382 字节。因此减小高速网络小数据包传输的延迟时间是提高网络系统性能的重要方法之一。

为了获得高带宽、低延迟的网络通信,必须对传统的通信协议作较大的修改,以克服传统协议的弊病,使高速网络的优越性能得以充分展现。

- 在用户空间实现通信协议

为了减少操作系统的额外开销,一个重要的方法是在用户空间实现一个用户态的协议层,使得此协议层能够旁路操作系统的影响,直接对网络硬件设备进行操作,这样就可减少数据拷贝次数,提高通信效率;把协议实现放在用户空间的另一优点是可以减少操作系统调用的时间开销,而且通讯协议能够与用户的实际应用密切结合,可减少协议不必要的冗余,同时也不会有损它的灵活性。不过,把通信协议放在用户空间实现,必须解决好两个问题:一个是多进程复用网络的问题;另一个是在没用核心参与的情况下,如何管理有限网络资源的问题。只有这样,用户态通信协议才能得以有效地实现。

- 精简通信协议

前面的分析说明,通信的开销很大程度上是由于协议层次多、数据拷贝频繁引起的,另外,通用的网络接口和协议为满足各种用户的需求,增加了很多与数据传输无关的服务,这些服务也带来了额外的开销。而在并行机群系统中,有些功能是不必要的,完全可以进行精简,以降低通信开销。所谓精简包括两部分内容:一部分是功能的精简,就是删除不必要和冗余的功能;第二部分是协议层次的精简,合并各层的功能,使得通信协议变为一层,以达到减少数据拷贝次数的目的。比如,在操作系统 Solaris 2.4 中,通信协议由网络驱动程序、数据链路层(DLPI)、IP 层、TCP 层和 Socket 接口组成,由于数据链路层 DLPI 已经提供了不保证数据包无差错传送的基本数据通信功能,因此可以在它基础上实现一个保证数据可靠传送的模块以取代复杂的 TCP/IP 协议层和 Socket 接口。这样新的通信协议不论从结构上看,还是从功能上分析,都比原有的协议要简单得多。精简协议的结构如图 9.31 所示。

- Active Message 通信机制

通信协议的用户态实现以及协议的精简这两种方法都是针对传统通信协议在实现方法上所做的改进,而 Active Message 则是一种全新的通信机制,能够更为有效地提高通信系统的性能。

A. Active Message 的原理

在 Active Message 通信方式下,消息的数据结构与传统的消息传递机制有所不同,它除了包含通常的数据项外,还增加了两项内容:消息处理程序指针 Handler 和参数。由于消息中包含消息处理程序指针,当消息到达目的结点后系统立即产生中断调用,并由中断处理机制启动消息处理程序。消息处理程序的功能是从网卡上取出消息并给发送方发送一个应答消息,然后返回原来被中断的程序。

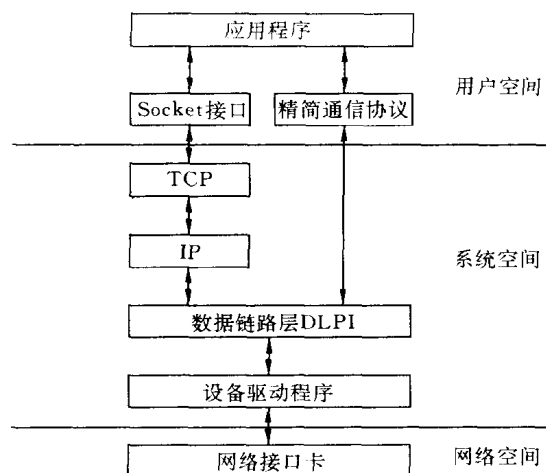


图 9.31 精简通信协议的结构

B. Active Message 机制的特点

1) Active Message 消息处理方式与网络硬件的处理方式相一致。对大多数网络硬件系统来说,当它接收到一个消息数据包后,立即会产生一个优先级很高的中断,通知操作系统消息已经到达,此时操作系统会调用相应的中断处理程序来接收并处理到达的消息。这是一个完全异步的消息处理过程。Active Message 正是直接使用了网络硬件提供的功能,让消息包的发送方预先指定好接收方用于处理该消息的函数,当消息数据包到达接收方时,这个预先指定的函数被调用来处理此消息。可见 Active Message 通信机制恰好顺应了网络硬件的通信过程,使得它能够更有效地发挥网络硬件的性能,有利于通信效率的提高。而传统的通信机制可实现同步通信和异步消息传输两种方式。前者通常采用“停-等”协议实现消息的可靠传输,与硬件的消息处理方式并不一致;后者则是通过引入复杂的缓冲管理机制来实现的,由于所有收到的消息首先要存放在缓冲器中,然后等待应用程序来处理,因此其软件处理开销较大,降低了通信系统的效率。研究表明,如果软、硬件操作模式之间的差异越大,则通信效率越低。使软件操作模式与网络硬件的消息处理方式相一致,以提高通信效率,即是提出这种 Active Message 通信机制的基本思想。

2) Active Message 通信机制是一种消息驱动的异步通信方式。异步通信最大的优点是能够实现通信与计算的重叠。与传统的异步通信方式相比,这种方式具有更大的优越性。其原因首先是消息驱动能使 CPU 获得较高的利用率;第二个关键原因是接收方收到的数据是由消息处理程序 Handler 提交给应用程序的,特别是当这种提交过程能由网卡上的处理器来完成时,就可使得 CPU 的计算与数据提交能够同时进行。

3) Active Message 通信机制能够简化缓冲管理。对数据接收方而言,由于在用户程序中已经预先分配了存储空间,接收到的数据可以直接存放那里,因此在接收方可以取消缓冲;对数据发送方而言,如果消息是大数据包,则需要把大数据包分割成若干块小的数据包放到缓冲中进行管理。如果消息是小数据包,由于 Active Message 通信处理小数据包的开销小,网络不易拥塞,使得网卡自身拥有的缓冲就足够用了,不需要另行分配缓

冲进行管理。

3. 几种典型系统

目前国内外许多科研机构都在对机群系统下的通信技术进行深入的研究,如 UCB (University of California, Berkeley)提出的 NOW 计划,Cornell 大学研制的 U-Net 系统,清华大学提出的精简通信协议 RCP 和快速消息传递机制 FMP 等。表 9.11 列举了在机群系统中实现的几种典型的通信子系统。

表 9.11 几种典型通信系统

系统名称	网络类型	实现技术	往返延迟时间 (μ s)	峰值带宽(Mb/s)
RCP(清华大学)	10Mbps Ethernet	精简通信协议	950 (TCP 为 1 438)	8.98 (TCP 为 8.92)
FMP(清华大学)	1.28Gbps Myrinet	快速消息传递	24.8 (TCP 为 220)	360 (TCP 为 252)
Fast Socket(UCB)	640Mbps Myrinet	精简通信协议	80	96
FM(UIUC)	640Mbps Myrinet	Active Message	50	130
AM(UCB)	640Mbps Myrinet	Active Message	31.5	152
U-Net(Cornell)	155Mbps ATM	Active Message	65 (TCP 为 1 285)	120 (TCP 为 80)
HPAM(UCB)	100Mbps FDDI	Active Message	29 (TCP 为 2 000)	96 (TCP 为 43)

从表中可看出,这六种系统虽然是在不同的网络硬件平台上实现的,但其性能无论是从延迟时间上看,还是从网络带宽上看,均比传统 TCP/IP 协议有较大幅度的提高,比如清华大学在 Myrinet 上实现的快速消息传递协议 FMP 通信延迟只有 24.8 微秒,比 TCP 的延迟时间降低了 8 倍,而网络带宽,达到 252 Mb/s,这说明通信协议软件处理开销的降低对网络性能的提高有着极其重要的作用。但由于网络速度不同,性能提高的幅度会有所不同,网络速度越高,则提高性能的潜力越大。RCP 和 FMP 分别在 Ethernet 和 Myrinet 上的延迟时间和峰值带宽数据清晰地说明了这一结论。

这些系统从实现技术上看,可分为两类,一类是采用精简通信协议的方法,另一类是使用 Active Message 通信机制。对比这两类系统的性能可知,采用 Active Message 通信机制实现的系统的性能比用精简通信协议实现的系统一般来说要好些。例如,UCB 在 Myrinet 上实现的 Fast Socket 的往返延迟时间为 80 微秒,而 AM 系统的往返延迟时间只有 31.5 微秒,可见,Active Message 通信机制能够更为有效地提高通信系统的性能。

高性能的工作站、高档微机的使用是提出并行机群系统的前提条件,而高速网络硬件设备的出现,则为机群系统的发展打下了坚实的基础,极大地推动了它的使用和推广,新型通信协议的研究又进一步发挥了高速网络的高性能,使得机群系统具有更好的性能和更广的适应性。因此可以预计,随着工作站本身性能的不断提高以及新的通信技术的研究,并行机群系统将会逐渐接近或达到 MPP 的性能指标,成为目前并行计算领域中的主流技术之一。

9.4.3.3 并行程序设计环境

近几年,并行处理技术发展很快,各种并行机已经广泛应用到了军事、石油、地震等多个应用领域。然而,由于并行应用程序开发设计上的复杂性,以及并行系统中应用软件的缺乏,使得并行程序的编写和调试变得相当困难,阻碍了并行计算机系统的进一步推广与应用。因此,为了适应并行处理技术的发展,充分发掘并行计算机系统中的冗余资源与计算能力,需要为用户提供一个良好的并行程序开发环境,提供多种层次的丰富的并行系统调用和工具包,使并行计算机系统的结构对用户透明,以减少并行应用程序开发设计上的复杂性。

广义地说,并行程序设计环境应包括硬件平台、操作系统和并行程序语言、编译、编程、调试及性能分析工具等,狭义的并行程序设计环境则仅指系统核心之上的工具软件部分。作为一个并行程序的支撑环境,至少应包括以下两个方面:

- 1) 并行语言支持或并行操作库函数支持;
- 2) 一种或多种并行编程模型。

并行计算机系统,依据其处理机间的连接结构,可以分为具有共享存储器的并行计算机系统和处理机间没有共享存储器的分布式并行计算机系统,像现在广为流行的机群系统即属于后一种。在分布式并行计算机系统中,由于处理机间没有共享内存支持,因而各处理机间通过消息传递机制来实现数据通讯,消息传递成为并行程序设计环境构造的基础。下面我们就对目前在分布存储多机系统以及机群系统上广泛使用的几个并行程序设计环境作一介绍。

1. PVM

PVM(parallel virtual machine)的开发最早开始于1989年夏天,它的开发队伍包括美国橡树岭国家实验室(ORNL)、Tennessee 大学、Emory 大学以及CMU等单位,并得到美国能源部、国家科学基金以及田纳西州的资助。PVM是一套并行计算工具软件,支持多种体系结构的计算机,像工作站、并行机以及向量机等,这些机器通过网络连起来,给用户提供一个功能强大的分布存储计算机系统。PVM支持C、C++和Fortran语言。由于它是免费的,因此使用范围非常广泛。

PVM支持用户采用消息传递方式编写并行程序,编程模型可以是SPMD或MPMD,计算以任务(task)为单位,一个任务通常就是一个Unix进程,每个任务都有一个taskid来标识(不同于进程号)。PVM支持在虚拟机中自动加载任务运行,任务间可以相互通讯以及同步。在PVM系统中,一个任务被加载到哪个结点上去运行,一般来说,对用户是透明的(PVM允许用户指定任务被加载的结点),这样就方便了用户编写并行程序。

PVM支持应用程序、机器以及网络级的异构性,允许应用任务选择最适合于它做计算的结点去运行。如果两台机器的数据表示格式不同,PVM在内部会自动解决数据转化的问题,PVM允许虚拟机采用多种不同的网络进行互连。

PVM系统中,各机器间的通讯基于TCP/IP,同时它也支持多处理机内部结点的通信直接利用多处理机本身提供的通讯函数,例如,Intel Paragon内部结点可直接利用NX消息通讯库进行通讯。

PVM 的特点可以总结为：

(1) PVM 系统支持多用户及多任务运行，多个用户可将系统配置成相互重叠的虚拟机，每个用户可同时执行多个应用程序。

(2) 系统提供了一组便于使用的通信原语，可实现一个任务向其它任务发消息，向多个任务发消息，以及阻塞和非阻塞收发消息等功能，用户编程与网络接口分离。系统还实现了通信缓冲区的动态管理机制，每个消息所需的缓冲区由 PVM 运行时动态申请，消息长度只受结点上可用存储空间的限制。

(3) PVM 支持进程组，可以把一些进程组成一个 group，一个进程可属于多个进程组，而且可以在执行时动态改变。

(4) 支持异构计算机联网构成并行虚拟计算机系统。

(5) 具有容错功能，当发现一个结点出故障时，PVM 会自动将之从虚拟机中删除。

PVM 系统由两部分组成，第一部分是 pvmd，它是一个 daemon 进程，驻留在构成虚拟机的每一台机器上，主要负责 PVM 系统的配置、用户任务的内部管理以及任务之间的通信等功能。第二部分是 PVM 用户接口 libpvm3.a，它包含了所有用户可使用的 PVM 库函数，像消息通讯、任务创建等，用户程序在编译时必须连接该库。

2. XPVM

XPVM 是 PVM 系统的一部分，它是 PVM 的图形控制台和监视器。XPVM 提供了一个 PVM 控制台命令和信息的图形接口和几个动态显示图，对 PVM 程序的执行进行监视，这些显示图提供了有关 PVM 应用程序任务间交互作用的信息，以帮助并行程序的调试和性能调整。

XPVM 是用 TCL/TK 以及 C 语言混合实现的，用 XPVM 对 PVM 并程序监视，可以不必重新编译应用程序。具体来说，XPVM 提供了以下功能：

(1) 配置 PVM 系统：可以动态地在 PVM 系统中增加或删除主机；

(2) 控制 PVM 任务：可以在 XPVM 中启动 PVM 程序运行，设置跟踪标志，还可以向 PVM 任务发送信号或者杀死 PVM 任务；

(3) 对 PVM 任务进行监视。

XPVM 提供了 5 种动态显示图。

网络显示图：网络显示图显示了主机在 PVM 系统上的高层次活动，每一台主机用一个图标表示，图标用不同颜色指示主机当前所处的状态。

空间-时间显示图：空间-时间显示图显示 PVM 系统中各个任务的状态及它们之间的相互通信情况。每个任务以沿着时间轴的水平条表示，水平条在每一时刻的颜色指示任务在该时刻的状态。“计算”表示任务正在执行计算工作；“开销”表示任务正在执行 PVM 函数调用；“等待”指明任务花在等待接收消息上的时间，任务中的通信活动也被两任务条之间的连线表示出来。

利用率显示图：利用率显示图显示了任一给定时间的处于计算、开销或等待消息的任务个数。

调用跟踪显示图：调用跟踪显示图提供每个任务瞬时活动的一个文字记录。对于每一个任务，显示的一行文字表示了该任务最近一次对 PVM 的函数调用，包括调用参数

和结果。

任务输出显示图：由于 TCL/TK 是解释执行的，因此用 XPVM 对 PVM 程序作跟踪时，速度较慢。新版的 XPVM 将很多以前用 TCL/TK 实现的部分改用 C 语言实现，在性能上得到了较大改进。

3. MPI

MPI(message passing interface)是一个新的消息传递标准，是由 MPI 委员会在 1992 年 11 月至 1994 年 1 月举行的一系列会议上逐渐产生的。该委员会由来自约 40 个研究院所和几乎所有的 MPP 销售商、以及世界范围内涉及并行计算的大学和政府实验室的成员组成。

MPI 能用于大多数并行计算机、计算机机群和异构网络环境，并达到较高的数据传输率，而且还同时具备了公共软件包和厂家专用软件包的优点。除此以外，由于 MPI 是经过审慎的设计，并建立在多种消息传递库的基础之上，所以能博采众长，扬长补短，使 MPI 的功能更加丰富和完善。

MPI 支持 C 和 Fortran 两种语言，编程模型采用 SPMD，它的编程要比 PVM 容易。MPI 系统的特点有：

(1) MPI 提供了缓冲区管理的函数，用户可以决定是完全由系统对发送、接收缓冲区进行管理，还是用户参与部分管理(向系统提交或释放自己的缓冲区)，以便更实际地控制系统的缓冲区空间，提高系统的安全性。

(2) MPI 能运行于异构的网络环境中，另外，MPI 还提供了一些结构和函数，允许用户构造自己的复杂数据类型，使得通信更加方便。

(3) MPI 通过通信上下文(context)提供通信的安全性，所有的通信都在一个上下文中进行。接收操作只能接收到同一上下文中发送的消息，即使别的上下文中任务在发送消息，它也不会去接收。同样，不同上下文中发送的消息也不会混淆。

(4) MPI 实现了两个任务间的多种通信方式，如阻塞式、非阻塞式通信，还有标准式、同步式、缓冲式和预备式发送操作。

(5) 集群通信(collective communication)：MPI 实现了组内所有任务之间的通信、数据交换及数据处理。在集群通信中，MPI 提供了丰富的数据操作函数，还允许用户定义自己的数据操作函数，使得通信和数据处理能更有效地结合起来。

(6) 错误处理：由于 MPI 提供了可靠的数据传输机制，发送的消息总能被对方正确地收到，用户不必检查传输错误、超时错误或其它错误条件。因此，MPI 未提供处理通信失败或处理机失败的机制，它只处理应用程序级的错误，MPI 能处理的每个错误都会产生一个 MPI 异常信息。

MPI 能基于的通信协议比较丰富，有的协议还不需要调用操作系统提供的通信功能，因此效率较高，另外，MPI 程序的执行不需要任何守护进程，其各个并行任务之间的消息传递是直接进行的，因此 MPI 比 PVM 的通信效率高。

4. EXPRESS

Express 系统是美国 Parasoft 公司推出的能在不同的硬件环境上运行的并行程序设计环境，它的前身是 Caltech 的 Crystalline 操作系统。Express 系统强调并行程序设计的

高层问题,它将许多与机器有关的细节问题都隐藏在 Express 函数中,目的在于更进一步为用户提供方便的并行程序开发环境。Express 支持 C 和 Fortran 两种程序设计语言。

Express 支持 Host-Node 与 Cubix 两种编程模型,每个模型对应于不同的并行程序库。在 Host-Node 模型中,用户首先编写在 Host 上运行的控制程序,然后再编写在结点上运行的结点程序。控制程序负责控制各结点程序的运行,同时完成回收数据及 I/O 等工作。

Cubix 编程模型由 Cubix 并行程序库提供支持。在这种方式下,应用程序完全分散在各结点上,在 Host 上由一通用的 I/O Server 来完成各结点程序间的协调及 I/O 服务工作,各结点可自由通讯,无须中央控制。

Express 提供了多种处理机间的通讯原语。除了通常的同步/异步通讯外,同时还提供了广播以及多重接收等功能函数,这些方式同时也适用于 I/O 服务。

在分配结点上,Express 可以对应用程序到并行计算机的映射提供帮助。对于 Cubix 方式,其结点分配是自动完成的。对 Host-Node 方式的映射,用户可以指定从哪个结点开始分配,也可以完全由系统自动完成分配。

对于系统的网络拓扑结构,Express 支持多种处理机间不同的连接方式,以适应不同的实际应用,但这一配置是静态的,须在应用程序运行之前进行配置。

Express 可以对系统及用户应用程序的运行做动态监控,它还提供了并行调试工具 NDB,可以帮助用户跟踪多道并行程序的执行以及它们之间的通讯。

另外,为了进行性能评测,Express 还提供了三种性能分析工具(对于 Host-Node 模式):

.ctool: 系统性能数据统计分析,包括

- 1) 计算时间,并行处理器结点间的通讯时间,执行输入/输出的时间;
- 2) 通讯系统遇到的出错次数;
- 3) 调用通讯函数的总次数。

.etool: 系统事件统计分析。

.xtool: 系统并行子任务执行行为统计分析。

Express 除了对以上传统的基于消息传递并行程序设计环境各方面作了许多工作以外,还对并行 I/O、动态负载平衡等方面提供了支持。

5. Linda

并行机一般可分为基于共享存储器的多处理机和基于分布存储器的多处理机。在共享存储多处理机上,一般采用共享存储的编程方式,它是对传统串行语言的扩充,编程较为容易;在分布存储多处理机上一般采用 message-passing 方式编写程序,像上面介绍的几个程序设计环境均属于此类。Linda 与前面介绍的编程环境不同,虽然它也是运行在分布存储多处理机上,但它引入了虚拟共享存储概念,通过在各处理机上实现一个虚拟的共享存储器,将分布存储多处理机模拟成为基于共享存储器的多处理机,从而达到支持共享存储编程方式的目的。

Linda 是美国耶鲁大学与科学计算协会共同研制的用于实现并行程序设计的、与机器无关的程序环境。Linda 通过增加一些函数对传统的程序设计语言进行扩充(如 C,

Fortran 等),使其能实现并程序的设计。将 Linda 映射到各计算语言中,就形成了可进行并行计算的程序语言 C-Linda 和 Fortran-Linda。Linda 可以运行在共享存储多处理机、分布存储多处理机以及工作站机群系统上。

Linda 的虚拟共享存储器称为元组空间(tuple space),元组空间由一组有序的元组(tuple)组成,元组的每个域都包含有实际的数据。元组空间是相联存储器,元组的标识与选择是通过域值匹配,而不是通常采用的地址选择方法。

Linda 提供了以下 4 种对元组的基本操作:

- out: 将数据放入元组空间,整个操作是顺序进行的;
- eval: 功能同 out,但它是并行执行的;
- in: 从元组空间中选匹配的数据,并将数据从元组空间中删除;
- rd: 功能同 in,但它不将数据从元组空间中删除。

当两个进程需要交换数据时,它们并不是采用 message-passing 来直接通讯,而是通过读写 Tuple 空间来完成(同共享存储多处理机的并行程序设计)。例如,如果 A 进程需要向 B 进程传递数据,则需要完成以下的操作:

- 1) A 向 Tuple 空间写一个数据;
- 2) B 在 Tuple 空间中读取需要的数据。

Linda 在编译系统和运行系统上作了很多优化工作,使用 Linda 编写的应用程序在效率上能够接近用传统 message-passing 编写的应用程序。

Linda 系统从其特点与功能上看,适合于处理一个结点上具有多个处理进程、进程间具有不明确的数据通讯、且进程间需要较强的同步机制以及全局通讯的实际应用问题。

6. IPCE

清华大学计算机系在承担的 863 项目“可扩展工作站机群系统”中,构造了一个由 8 台 Power PC 工作站,采用 100Mbps 以太网连接起来的机群系统。系统采用 PVM 作为低层编程环境,高层以 XPVM 为基础。通过对其改进与扩充,开发了一个高效实用的可视化集成开发环境 IPCE。

IPCE 是一组图形界面工具,它使得程序员可在多机环境下实时地监视和控制同时在不同机器上运行的程序。它在 XPVM 的基础上增加了编辑、编译链接装配、各结点资源使用情况显示、并行调试以及地震数据处理接口等功能。另外,IPCE 还扩充了 XPVM 的动态显示图,增加了显示各任务发送/接收消息的字节数,各结点发送/接收消息次数以及字节数等状态监视图。总之,IPCE 为用户使用并行机提供了方便。

9.4.3.4 机群系统负载平衡技术

1. 负载平衡的基本问题

在并行处理系统中,一个大的任务往往由多个子任务组成。这些子任务被分配到各个处理结点上并行执行,称之为负载。对于由异构处理结点构成的并行系统而言,由于各结点的处理能力不同,相同的负载在其上运行的时间和资源占有率都不同。因此,准确的负载定义应是绝对的负载量与结点处理能力的比值。当整个系统任务较多时,各结点上的负

载可能产生不均衡现象,就会降低整个系统的利用率。这就是负载不平衡问题。负载不平衡问题解决得好坏,直接影响到系统性能,因此它就成为并行处理中的一个重要问题。

为了充分利用高度并行的系统资源,提高整个系统的吞吐率,就需要负载平衡技术的支持。负载平衡技术的核心就是调度算法,即将各个任务比较均衡地分布到不同的处理结点并行计算,从而使各结点的利用率达到最大。

负载平衡系统的一般评价标准包括:

(1) 吞吐率(throughput):并行系统上运行的应用程序的响应时间或平均完成时间。这是负载平衡系统最主要的衡量尺度。

(2) 可扩展性(scalability):系统规模增大或总负载大小变化时系统的适应能力。

(3) 容错性(fault-tolerant):发生处理机故障后任务恢复运行的能力。

在机群系统上,负载平衡要解决的问题主要有以下几点:

(1) 系统资源使用不均。以 CPU 资源为例,在并行计算中常常会出现这样的现象,某个结点的 CPU 处于十分繁忙的状态,而另一些结点却非常空闲。这导致了系统资源的极大浪费和并行效率的下降。在机群系统中,由于现有的并行编程/运行环境负载平衡机制通常十分简单,同时各个结点的体系结构和资源状况互不相同,如何充分利用资源和调度负载就主要成为程序员的责任。但并行程序运行时系统资源状况在动态改变,程序员很难作出准确的并行任务静态调度。因此,必须动态监视系统的资源状况,作出准确的分配决策。

(2) 机群系统是多用户系统,同时可能有几个用户运行各自的作业。这就要保证前台用户对工作站的优先使用权。而现有并行环境对此考虑不多,以 PVM 为例,它采用 Round Robin 任务分配策略,并行任务依次派生到各个结点上,不管该结点是否被前台用户使用。因此,必须在某前台用户频繁操作时尽量减少分配给该结点的任务。这样做既能使前台用户不会因后台任务太多而无法忍受过长的响应时间,又能充分利用空闲结点的资源。

2. 负载平衡系统的主要技术

(1) 负载平衡决策时机

从任务分配决策时机的角度出发,负载平衡技术可分为静态和动态两大类。所谓静态方法就是在编译时针对用户程序中的各种信息(如各个任务的计算量大小、依赖关系和通信关系等)以及并行系统本身的状况(如网络结构,各处理结点计算能力等)对用户程序中的并行任务作出静态分配决策;而在运行该程序的过程中依照静态分配方案将任务分配到相应结点。理论证明,静态算法求最优调度方案属于 NP-Complete 问题,因此在实践中往往采用求次优解的算法。静态算法要求获知完整的任务依赖关系信息,但在高度并行的多计算机领域,特别是在多用户方式下,各处理机上的任务负载是动态产生的,不可能作出准确的预测。因此,静态负载平衡方法多用作理论研究和辅助工具。

动态负载平衡方法则是在应用程序运行过程中实现负载平衡的。它通过分析并行系统的实时负载信息,动态地将任务在各处理机之间进行分配和调整,以消除系统中负载分布的不均匀性。由于各处理机上的任务是动态产生的,因此在程序执行期间,某台处理机上的负载就可能突发性地增加或减少。这时,重载(负载较重)的处理机应及时地把多余的

任务分配到轻载的处理机上去(或者轻载的处理机及时地向重载的处理机申请任务)。动态负载均衡的特点是算法简单,实时控制,但同时也增加了系统的额外开销。

(2) 调度系统模式

动态调度系统模式可分为集中式和分布式。

集中式负载均衡控制一般通过 Master/Slave 的方式来实现,最典型的是任务池(pool of tasks)方法。该方法由 Master 创建和维护任务池(一般组织成队列形式),并向空闲的 Slave 处理机分派任务。如果每个任务粒度不同,则粒度较大的任务一般处于队列头部。在 Master/Slave 方式的负载均衡控制中,各 Slave 只服从 Master 的命令,互相间不进行通信;另一种比较典型的集中式负载均衡方法是换维平衡法(dimension exchange method),该方法是一种全局控制的、完全同步的策略。它首先将 N 个处理机组织成 $\log(N)$ 维,然后每次在某一维的处理机集合中进行负载均衡。通过更换不同维的处理机集合,利用维与维之间处理机的重叠特性,达到全局负载均衡的效果。集中式负载均衡在大规模并行处理系统中可能会出现系统瓶颈现象。

在分布式负载均衡控制中,各处理机分别进行调度。它又可分为合作型以及非合作型两类。前者的调度决策基于整个系统状况,准确度高但网络开销大;而后者的调度决策则基于局部信息,相对而言网络开销小,但决策准确率低。所以,实际的系统实现往往是这两种类型的折衷。分布式控制系统不会因一个结点的故障而崩溃,故健壮性较强。

(3) 负载指标的设计与收集

动态调度中需要获得各结点负载信息,这包括 CPU 处理能力、CPU 利用率、CPU 就绪队列长度和进程响应时间等。CPU 处理能力反映的是不同类型的处理机计算能力的强弱。CPU 利用率定义为单位时间内 CPU 处理用户进程与处理核心进程的时间比。当 CPU 利用率很低时,可以认为 CPU 处于空闲状态;当 CPU 利用率接近 100% 时,采用 CPU 就绪队列长度衡量负载轻重。由于 UNIX 系统是有优先级的固定时间片分时系统,故还可采用测试特定进程响应时间的方法来估计系统负载。另外,磁盘可用空间、内存,以及 I/O 利用率也作为一项重要的负载指标。

在集中式负载均衡控制中,各结点收集本地负载信息,并以一定时间间隔向控制结点报告。这里时间间隔的设置对性能影响很大:太短会引起通讯拥挤,太长则影响调度的准确性。在分布式控制中,各结点也必须收集本地负载信息,在信息交换时可以有两种选择,既可以定时交换,又可以只在发生任务调度时交换。

(4) 负载调度策略

负载调度策略的作用是决定调度由谁发起,和由谁接受任务。常用的方式包括直接传递(direct)、全局信息式(focused addressing)和竞争式(bidding)。

在直接传递方式下,各结点根据本身及相邻结点信息发出任务或申请任务。它还包括派生者发起(SI)和接受者发起(RI)两种类型。SI(sender-initiated)策略利用处理机邻域的负载信息,由重载处理机主动发起负载均衡,将过多的负载送到邻域中的轻载处理机上,因此它是一种高度分布的方案。在 SI 系统中,全局负载均衡通过重载区域向轻载区域扩散负载来实现。这里有两个主要问题:如何确定一个处理机负载已达到阈值和如何选择任务要送到的结点。SI 方式在系统平均负载较轻的情况下更有效。RI(receiver-initiated)

方式又称为选拔法。在 RI 系统中,负载平衡由轻载处理机主动发起申请,邻域中的重载处理机在收到该申请后,将适量的负载传递给轻载处理机。由于轻载处理机承担了绝大部分负载平衡的开销,因此 RI 方法在系统平均负载较重的情况下性能较好。RI 方式采用重叠域思想,是目前分布式负载控制技术中,特别适合于支持大规模并行系统解决各种应用问题的优选方法。

在全局信息方式下,每个结点都保留一份全局负载信息表,在任务派生时将任务直接发送给最轻载的处理机。

竞争方式则是在结点发生任务派生时向各个结点发出申请,各个结点根据自身负载情况进行竞争,最终将任务发送给负载最轻的结点。

3. 负载平衡系统设计的考虑因素

下面我们对机群系统中的网络、机器结构、系统利用率、可靠性以及编程模式等方面对负载平衡系统设计的影响作一个分析。

(1) 网络

当前,常用的局域网络技术有两类。一类是共享介质网络,最常见的是用 10/100 Mbps 的 Ethernet 连成的总线结构的系统,其价格便宜。另一类是开关网络,如 155Mbps ATM,其性能优异,但价格很高。对于共享介质网络,结点之间点到点通信要占用整个传输介质。而实现广播的开销则与点到点相同。这就是说,采用集中式负载平衡系统(每个结点定时向调度结点发送负载信息)与采用合作式分布负载平衡系统(每个结点定时向其它结点广播自己的负载信息)的网络开销相当,因此在选择负载平衡系统时要考虑的问题主要是由于网络冲突可能性带来的网络性能下降,解决的途径是合理调整结点间的负载信息交换时机(如定时通讯或调度时通讯)。对于开关网络,采用集中式负载平衡系统会产生通讯瓶颈(与调度结点频繁通讯),并直接影响到并行系统的可扩展性,所以通常选择非合作式分布系统,这时要考虑的问题在于如何利用不完整的局域负载信息在可以容忍的时间内达到整个并行系统的负载平衡,以及如何避免负载颠簸(thrashing)。

(2) 机器结构

机群系统分为同构(homogeneity)和异构(heterogeneity)两类。同构机群系统各结点为运行同一操作系统的同一型号的主机,且内部配置相同。各结点从处理能力、数据内部表示法到内核映象等都是一致的,这给资源比较、负载评价、数据传递、进程迁移的实现带来了方便。但在实际环境中,大多数机群系统都是异构机群系统,这就要求并行计算环境不仅要能够准确地评价各结点的处理能力、负载状况,还要求提供正确的数据通信机制,完成不同数据表示法之间透明的转化。对异构机群系统来说,任务迁移是很难实现的,因为不能像在同构机之间那样将内核映象直接复制,而必须采用更高层次上的迁移,即源代码在目标机上重新编译,并将环境恢复到迁移点的状态。

(3) 系统利用状况

指使用机群系统的用户数量、系统的平均负载量、负载方差及通讯量等状况。系统利用率高、平均负载重、网络的响应时间长、通讯量大,则网络拥挤,易造成并行任务间的同步等待时间长。负载方差大说明负载分布不均衡,负载平衡系统对性能影响就大。另外,由于机群系统是多用户系统,负载平衡系统应保证各用户的任务响应时间不能太长,这意

味着对于负载较重的结点,应尽量减少负载平衡系统本身带来的开销。

(4) 系统的可靠性

系统内工作站结点的软、硬件故障以及网络故障均会导致系统部分结点失效,不仅使系统功能下降,更重要的是正在进行的计算任务将会失败。这要求负载平衡系统提供容错机制。首先是系统本身不能因为故障而崩溃,要做到这一点,一个分布式系统远比集中式系统要好。其次,要能迅速发现并登记故障,以避免故障结点对正常结点的影响,且不再向故障结点分配任务。一般采取的方法是超时判定法,即当在一定时间内不能收到某结点发的消息,则认为该结点失效。需要指出的是:这种机制在集中式系统中实现较为简单,因为它有唯一的一个管理结点可以直接判断出超时的结点并修改系统配置;而在分布管理的系统中则不易实现,需要结点间经过协商才能确定一个失效结点。因此,究竟采用分布式还是集中式系统,还要视具体情况而定。

(5) 编程模型

机群系统并行计算编程模型分为共享存储模式(shared memory)和消息传递模式(message-passing)两类。共享存储模式的典型实现如 Linda 和 CSL(calypso source language): Linda 通过元组操作来提供并行任务间的信息交换,CSL 则向用户提供一组并发原语。在共享存储模式中,任务划分由编译器完成。因此,负载平衡的实现大部分在编译时完成。使用消息传递模式的实用系统有 P4、PVM、Express 和 MPI 等,其中 PVM 比较典型。它是一个在异构型网络环境中模拟一个通用的分布式存储多处理机的软件系统,提供一个基于消息传递的、对用户透明的并行编程运转环境。消息传递模式由于其灵活性及实用性在并行计算领域被广泛采用。对于消息传递模式,负载平衡系统独立于用户程序,主要是通过对运行时系统的扩充来实现的。

4. 机群系统负载平衡技术的研究状况

(1) 现有系统介绍

美国 Wisconsin-Madison 大学的 CONDOR 系统可以在大多数 UNIX 平台工作站机群系统上运行。它的设计目的是调度长运行时间的计算作业充分利用空闲工作站进行计算,并保证当空闲工作站的交互式用户开始使用机器时及时撤走它调度的后台任务。CONDOR 不需要重新编译用户程序,但需要把它们重新链接。另外,CONDOR 研究小组还开发了一个系统接口(CARMI),将并行环境 PVM 和 CONDOR 结合起来实现任务级负载平衡。

加拿大 Platform 公司的 LSF 系统是一个商业化软件。LSF 可以看成是一个通用型分布计算系统。它通过对资源的较好利用,把机群系统集成成一个对用户来说是单一的系统。LSF 支持串行或并行政程序的交互式或批处理式运行,并提供 C 语言函数库,使用户能够在编写并行程序时利用 LSF 的系统功能。LSF 核心由 LIM 和 RES 两部分组成。其中 LIM(load information manager)的功能是负责收集本地负载信息,向 Master LIM 报告,由 Master LIM 对作业进行分配,并通知相应机器的 RES(remote execution server)执行。LSF 具有对作业级的任务进行透明的集中式管理、各结点的 RES 接受远程执行请求并提供透明执行等特点。

我国吉林大学的 ILBOT 系统是用于工作站机群系统的智能动态负载平衡软件,它

具有使用不同的资源指标来衡量不同类型的作业对资源的需求、使用在线跟踪技术进行作业选择等特点。

美国 Arizona State 大学的 CALYPSO 是一个开发程序中的并行性的并行系统,它向用户提供了一个扩展 C++ 编程语言 CSL(calypso source language),支持 DSM(distributed shared memory)编程模式,相应增加了并行操作原语:shared,parbegin,parend 和 routine。运行时系统进行动态负载平衡,并具有容错能力。Calypso 系统分离了逻辑并行性和物理并行性,使用户无需考虑数据划分和任务分配。

(2) 需进一步研究的问题

1) 加强对任务级并行粒度的支持

独立的负载平衡系统往往只支持粗粒度并行,如作业级并行,而缺乏对并行程序在运行中产生的大量并行任务的调度功能。因此目前对任务级的负载平衡依赖于并行计算系统本身。对于像 CALYPSO 这类并行系统,由于采用对用户透明的 DSM 模式,本身具有很强的负载平衡机制。然而大多数基于消息传递模式的并行环境如 PVM 等,因为并行任务的划分由程序员完成,所以基本上没有负载平衡机制。MPI 标准也未制定负载平衡的接口。目前 LSF 和 CONDOR 的研究人员正在开发相应的任务级负载平衡功能,也有一些研究单位进行独立的任务级负载平衡系统的研究。

2) 提高任务分配的准确性

动态任务分配虽然具有很高的灵活性,但由于无法对即将派生的任务的性质,诸如任务粒度、同步关系和通信量等作出预测,从而无法准确地进行任务分配。因此某些研究尝试采用神经元学习等预测机制,提高分配准确度。

3) 减小额外的网络开销和计算开销

负载平衡系统占用网络资源过多,如 LSF,其对服务器结点的共享采用 NFS 网络文件系统,从而带来了额外的网络开销。减小开销的途径之一是合理调整负载平衡系统的信息交换时机,之二是采用局部信息进行分布式调度。后者也是减少调度计算开销的途径之一。

习 题 九

9.1 解释下列术语

共享存储多处理机;分布存储多处理机;SMP;S2MP;MPP;机群系统;
虚拟共享存储器;Cache 一致性;监听协议;写一次协议;基于目录协议

9.2 多处理机在结构、程序并行性、算法、进程同步、资源分配和调度上与并行处理机有什么差别?其根本原因是什么?

9.3 多处理机有哪些基本特点?发展这种系统的主要目的有哪些?多处理机着重解决哪些技术问题?

9.4 画出多处理机两种不同结构的框图。从哪些途径可以减少多处理机访问主存的冲突?

9.5 分析并行处理机、单处理机流水方式、多处理和单处理机“分析”“执行”一次重叠方

式这 4 种系统,分别属何种指令流/数据流系统?表现出何种并行性?能达到何种并行性等级?各遵循何种并行性途径发展而来?影响系统吞吐率和效率提高的主要问题是什么?如进行向量运算 $C=A+B$,向量长度较长时,它们连同单处理机顺序方式相比,比较在吞吐率、设备量、设备利用率方面的优劣。

- 9.6 假设有两个处理器,处理器甲的速度是乙的两倍,参考性能模型公式 9.1,请问如何分配任务以达到最优性能?
- 9.7 性能模型公式 9.2 所表示的模型适用于传送时间与处理器个数无关的系统。一次通信的开销为常数 C ,总开销等于 C 与通信次数的乘积。而在一个令牌环(token ring)内,传送时间与处理器的个数成正比。设计一个能体现令牌环这种特性的模型,并为你的模型设计一个优化的任务分配算法。
- 9.8 本题的目的在于设计一个与实际程序相符的性能模型。参考程序 6.1,最内部的一对循环更新矩阵内的一个矩形区域,而外层循环重复该操作 n 次。回答下列问题(忽略进程同步和计数的开销,只考虑数据通信开销):
 - ① 分配此矩阵,使每一行由一个处理器处理,请确定这种算法中必定要进行的处理器之间数据传送情况。如果无广播机制,那么此算法中要进行多少次数据传输?并与串行计算机与多处理器的计算机相比较。
 - ② 如果所设计的体系结构支持单周期的广播机制,此种机制能使一个处理器在一个周期内向所有的接收处理器发送一个消息。在此前提下重复①。
 - ③ 已知 $N=10$, $R/C=1$,在具有广播机制的系统中,如何向处理器分配任务以取得最优效果?
- 9.9 重复题 9.8,条件改为矩阵的每一列存储于一个处理器中,与每一行存储于一个处理器的方案进行比较并讨论存储格式对任务最优分配的影响。
- 9.10 本题的目的是研究同步对系统的影响。对于题 9.8 中的列方向数据结构,重新考察程序 6.1,并指出哪里需要同步。修改题 9.8 的性能模型以便计算出所需的同步操作个数。
- 9.11 假设程序 6.1 中的矩阵存于 N 个处理器上,每个处理器上有一列。当子数组更新时,每列同时更新。更新结束时,可通过驻留在 0 号处理器上的共享信号量来进行同步。在一次循环开始时,这个量被初始化为参与循环过程的处理器个数。当某个处理器完成其任务后,就独占这一信号量,将其减一,再释放。当一处理器将信号量减为 0 时,它就开始启动更新下一个子数组。否则,进行减一操作后就处于空闲状态。当 $N=16, 32$ 和 128 时,对于以交叉开关互连的多处理器求在等式 7.1 中的参数 r 和 h 的值。
- 9.12 程序 6.1 的结构要求对矩阵的行与列均进行访问。考虑一个通过二次扫描访问矩阵的简单算法,第一次访问矩阵的行,第二次访问矩阵的列,矩阵规模为 $N \times N$ 。
 - ① 对于一个基于交叉开关、具有 N 个处理器的多处理器系统,指出如何存储矩阵使得按所需形式访问矩阵时间最小,且说明完成两次扫描需要多少时间。
 - ② 对于总线型的多处理器系统,重复①。
- 9.13 解决多处理机系统中的多 Cache 一致性问题可采用哪些方法?叙述它们的优缺点。

- 9.14 何谓大规模并行处理机？它的主要特点是什么？
 9.15 何谓 SMP？它的主要特点是什么？
 9.16 何谓机群系统？它的主要特点是什么？
 9.17 何谓虚拟共享存储器？叙述它的优点和实现方法。
 9.18 试确定在下列 4 种计算机系统中，计算下列表达式所用时间。

$$S = \prod_{i=1}^8 (A_i + B_i)$$

其中，加法需用 30ns，乘法需用 50ns。在 SIMD 和 MIMD 计算机中，数据由一个 PE(处理单元)传送到另一个 PE 需要 10ns，而在 SISD 计算机中数据传送时间可忽略不计。在 SIMD 计算机中 PE 间以线性环方式互连(以单向方式传送数据)，而在 MIMD 计算机中，PE 间以全互连方式连接。

- ① 具有一个通用 PE 的 SISD 计算机系统；
 - ② 具有一个加法器和一个乘法器的多功能部件的 SISD 计算机系统；
 - ③ 具有 8 个 PE 的 SIMD 计算机系统；
 - ④ 具有 8 个 PE 的 MIMD 计算机系统。
- 9.19 分别确定在下列各计算机系统中，计算点积 $S = \sum_{i=1}^8 (a_i \times b_i)$ 所需的时间：

- ① 通用 PE 的串行 SISD 系统；
- ② 具有一个加法器和乘法器的多功能并行流水 SISD 系统；
- ③ 有 8 个处理器的 SIMD 系统；
- ④ 有 8 个处理机的 MIMD 系统。

设访存取指和取数的时间可以忽略不计；加与乘分别需要 2 拍和 4 拍；在 SIMD 和 MIMD 系统中处理器(机)之间每进行一次数据传送的时间为 1 拍，而在 SISD 的串行或流水系统中都可忽略；在 SIMD 系统中，PE 之间采用线性环形互连拓扑，即每个 PE 与其左右两个相邻的 PE 直接相连，而在 MIMD 中每个 PE 都可以和其它 PE 有直接的通路。

第十章 多处理机算法

本章研究高性能多处理机的程序设计方法,其中很大一部分内容是研究各种有效的能确保程序正确地执行的机制。首先研究简单并行性,它在许多应用领域已获得巨大成功。

要想通过并行处理获得最高性能,必须探讨更深入的概念。本章举了一个搜索算法例子。如果程序员想当然地把互相关联的任务分配给不同的处理机,那么所能获得的加速比是很低的。在这种情况下,搜索空间分配给了所有处理机,当任何一台处理机找到一个解时,搜索就结束了。

本章介绍一种与众不同的解决“旅游商问题”的优化方法,本方法非常有效地利用了并行性。采用这种方法使解决该问题的平均时间的增长小于问题规模的二次方的增长。这是非常好的结果,因为“旅游商问题”属于完全 NP 问题,是公认的难题之一。

因此直到现在还没有一种算法使解决该问题的时间的增长小于问题规模的指数的增长。以往理论上只考虑最坏的情况而没有涉及平均情况,这里我们将讨论复杂度很低的平均情况。

为了保证并行算法的正确性,需要有一种对共享变量进行修改的机制。这里引入一个能反映性能的参数 SYPS(synchronizations per second),它表示每秒钟同步的次数,常以 MSYPS(兆 SYPS)为单位。

本章将讨论 MSYPS 对吞吐率的影响。吞吐率既受 MIPS 的限制又受 MSYPS 的限制,它不能超过这两个参数所允许的最大值。一台高 MIPS 低 MSYPS 的机器在执行数值运算时可能非常高效,而处理需要大量同步的应用问题则可能很低效。我们不能孤立地考虑 MIPS 或 MSYPS,需要把两者结合起来考虑。虽然孤立的 MIPS 值表示有很高的吞吐量,但系统结构的另一个参数 MSYPS 可能将阻碍 MIPS 潜在能力的发挥。

10.1 简单并行性

并行性在需要很多次循环的程序中最有用。如果能把一个程序的执行时间由十天减少到一天,则一般认为这是很值得做的事情。如果把一个程序的执行时间由十分钟减少到一分钟,那么虽然减少的百分比与前者相同,但却不那么令人感兴趣。我们可以肯定较长的程序几乎一定包含一些需要重复执行许多次的程序段,这些段的执行时间占整个程序执行时间的绝大部分。

设一台计算机的时钟周期为 100ns,则一天有 10^{12} 个时钟周期。假设有一个需要运行一天的程序,那么其大部分运行时间花在什么地方呢?如果某个子程序或某指令序列需要重复执行许多次,比如说一百万次,那么我们的论点得以证明了。

设每条指令需要十个时钟周期,并且重复执行十次,我们将发现一个需要运行一整天

的程序必须包含 10^{10} 条不同的指令。这一程序就其规模来说是不平常的,要编制这样的程序,按当前的软件编制能力,需要上千人年。这一程序很可能只有 $10^4 \sim 10^6$ 条指令,所以平均重复率大约为 $10^5 \sim 10^7$ 。

假设有一段需重复一百万次或更多次的指令序列,如果能把这一百万次执行通过某种方法分配给 N 台处理机,我们就获得了并行性。下面是获得并行性的简单步骤:

- (1) 分析程序的循环或递归结构。
- (2) 找出执行时间最长的指令序列。通常是循环次数最多的程序段。
- (3) 在保证正确性的前提下,把这些程序段分配给 N 台处理机去执行。
- (4) 为保证并行执行的正确性,增加一些同步和数据传输语句。

程序 10.1 是运用上述方法的一个实例。该程序实现第八章介绍过的泊松计算。回顾我们以前的讨论,当时我们认为近邻迭代常常不是解决泊松问题最有效的方法,然而程序 10.1 还是用了迭代方法。

程序 10.1 串行地求解泊松问题

```
for K := 1 to 10 × M do
begin
  for i := 1 to M do
  begin
    for j := 1 to M do
    begin
       $P[i, j] := (P[i, j+1] + P[i, j-1] + P[i+1, j] + P[i-1, j]) / 4;$ 
    end j loop;
  end i loop;
end K loop;
```

假设我们事先已经知道达到收敛需要 $10M$ 次循环。程序 10.1 中有三重循环。最外层循环重复 $10M$ 次后达到收敛。这里把外层循环次数固定,只是为了简化例子。许多实际情况是要重复执行最外层循环,直到收敛的结果满足为止。

经过第二层和第三层循环后,长方形内每一点 $P[i, j]$ 的值均被修改一次。最内层循环操作结束时某列的 M 个点的值被修改了一遍。中间一层循环操作结束时把长方形的所有列的点都修改了一遍。最外层循环使长方形的各点的值被修改 $10M$ 次。

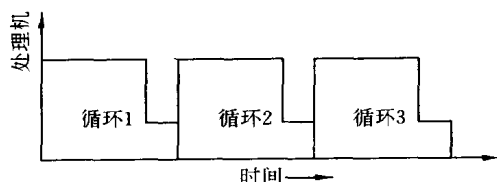


图 10.1 处于忙状态的处理机数目与时间的函数关系

一个完全是串行的程序在把长方形内的所有点都修改一次后才能开始下一轮修改。为了使并行程序也按这种顺序修改,需要对并行性加以限制,只对长方形内各点的一次修改并行地执行。这种并行的修改一次接一次地执行 $10M$ 次,它们之间不出现重叠。图 10.1 是一种可能出现的执行情况。表示了处于忙状态的处理

机数目和时间的函数关系,图中循环 1、循环 2、…是指外循环。在一次循环的大部分工作

完成之前,所有可用的处理机均处于忙状态。当一次循环快要完成时,一些处理机空闲下来,必须等待到下一次循环开始才能重新计算。

10.1.1 do par 和 do seq 结构

从程序设计的角度来看,我们需要把并行与串行的概念引入程序设计语言中,以便区分能在多台处理机上并行执行的循环还是必须一次接一次串行执行的循环。把如下形式的 do 结构引入 Pascal 或 Fortran 语言是扩展这种语言的最简单方法。如:

(1) do par 表示并行执行循环;

(2) do seq 表示串行执行循环。

因此,

```
for i := 1 to M do seq
  begin
    iteration A
  end do seq;
```

这个程序串行地执行 A 循环体 M 次。

```
for i := 1 to M do Par
  begin
    iteration A
  end do par;
```

这个程序同时激活 A 循环体的 M 个副本,每个副本中的 i 值是不相同的,根据调度算法和可用资源的情况,全部或部分地执行这些副本。

下面我们使用 do par 和 do seq 结构把程序 10.1 改写成程序 10.2。在程序 10.2 中,两个内循环用了 do par 结构,外循环用的是 do seq 结构。程序运行过程中,生成 M^2 个内循环副本,每个副本和一个 (i, j) 相对应,并把这些副本分配到多台处理机上执行,然后等待各处理机执行的结束。当各处理机都执行结束时,程序重新执行上述这个过程,这个过程重复执行 10M 次。

程序 10.2 并行地求解泊松问题

```
for k := 1 to 10 × M do seq
  begin
    for i := 1 to M do par
      begin
        for j := 1 to M do par
          begin
             $P[i, j] := (P[i, j+1] + P[i, j-1] + P[i+1, j] + P[i-1, j]) / 4,$ 
          end j Loop;
        end i Loop;
      end K loop;
```

10.1.2 阻塞同步

程序 10.2 的 do seq 结构隐含着同步信号。一台处理机只有在被告知上一次外层循环全部完成以后才能开始下一次外层循环。

事实上,do seq 结构已经在每次循环结束处设置了一个阻塞。do seq 循环可以分配到尽可能多的处理机去执行,但这些处理机必须在每次循环结束的阻塞处停下来。任何一台处理机在所有执行循环体的处理机到达阻塞之前不能越过这个阻塞。

在程序 10.2 中,我们可以用多达 M^2 台处理机同时执行一次外循环,但先到阻塞的那些处理机必须停下来等待,直到所有处理机都到达阻塞才能开始执行下一次循环。这种同步称作阻塞同步。尽管在程序 10.2 中没有专门的同步语句,但在 do seq 的每次循环末尾隐含了一个同步。

程序 10.3 的 do par 结构循环中使用了显式阻塞同步。循环体由 Step A, Step B 和 Step C 组成。do par 结构生成 M 个循环体,每一个对应一个 i 值,然后把这些循环体分配给所有可用的处理机。

程序 10.3 显式阻塞同步的例子

```
for i := 1 to M do par
  begin
    Step A(i)
    Step B(i)
    Barrier;
    Step C(i)
  end i Loop;
```

注释: Barrier 语句使循环 C 在循环 A 和循环 B 全部完成后才能开始执行。

如果不使用阻塞同步语句,对每一个 i 值来说,我们必须保证按 Step A(i), Step B(i), Step C(i)的顺序执行。在不同 i 值的循环之间,Step 的顺序则是随机的,什么情况都可能出现。例如,可能出现下面这种执行顺序: A(1), A(2), B(2), C(2), B(1), C(1)。因为对某一 i 值来说,循环体内部是顺序执行的,所以 B(1)在 C(1)之后执行的情况根本不会发生。

程序 10.3 在 Step B 之后加了一个阻塞同步语句,其作用是迫使所有的循环都执行完 Step A 和 Step B 以后再开始执行 Step C。有了阻塞同步语句后, A(1), A(2), B(2), C(2), B(1), C(1)这种执行顺序不会出现,因为 C(2)不可能在 B(1)之前执行。如果每次循环的 Step C 必须依赖于以前各次循环的 Step A 和 Step B,则必须在 Step C 之前设置阻塞。

阻塞同步是一种有效的方法。我们还可以采用更复杂的同步方法,使得不同循环中的 Step C 能早一些开始。这种方法必须有判断能力,当它发现某些特定条件成立时,便开始执行 Step C。这种判断能力比仅仅判断所有处理机都到达阻塞这一种情况要复杂得多。

10.1.3 性能分析

有了并行执行循环和串行执行循环这些概念之后,再来考查一下并行程序的性能。先忽略诸如在 do par 开始时进行并行任务的初始化、阻塞处理等细节问题。主要目的是求出程序的 R/C 比值,这样就可以利用第九章的结论来改进多处理机的算法。

程序 10.2 的内层循环的一条语句是一个任务。这条语句大约需要 6 条指令,它们是一条 LOAD,三条 ADD,一条 SHIFT 或 DIVIDE,一条 STORE。同时还需要地址计算。但如果地址计算可全部由地址变换机构完成,不需要额外的指令的话,那么地址计算的时间可忽略不计。另外还需要为每次循环计算 i 和 j 的时间。这样,执行一次循环大约要执行 10 条指令。这相当于程序的运行时间 R 。

用 C 来表示额外开销和通信所需的时间,它包括生成任务、当所有处理机都处于忙时将任务进入队列、当处理机可用时调一任务出队以及为了保证在所有任务全部完成后能通过阻塞还需记录任务的完成情况。

有时候我们可以幸运地避免入队和出队操作,但必须有一些指令来完成任务的生成和终止。生成任务和终止任务的最小开销是各需要两条指令,实际的开销可能是几百甚至几千条指令。这样, R/C 比值可能高达 2 或 3,也可能低到 $1/100$ 或 $1/1000$ 。对第九章所讨论的大多数模型来说,这种比值不可能支持很高的并行性。根据机器的系统结构和 R/C 比值,程序 10.2 在一台或少量几台处理机上执行的速度可能最快。这还是比较乐观的结论,因为以前所讨论的模型都忽略了同步的影响。如果考虑到同步的影响,处理机的台数可能还要少一些。

以程序 10.2 为例,让我们看一下同步对单个任务的影响。对每一个任务来说,要生成它并要入队、出队、终止处理等。入队、出队和终止处理的过程中很可能要涉及对共享变量的修改。若任务的生成过程中要修改共享变量,那么也会增加总开销。

我们把修改共享变量作为一个基本操作,称为 Synchronizing (Synch)。这样,程序 10.2 中的一个任务需要 3 个 Synch (分别用于入队、出队和终止处理)和大约 10 条指令(用于任务生成、循环体执行和任务终止)。Synch 操作实现方法对大多数多处理机系统结构影响很大。可并行地执行的 Synch 数量相当有限,有时甚至到只有一个。但具有合并开关的系统结构如 IBM 的 RP3 和 NYU 的 Ultra 计算机例外。

为了更透彻地理解 Synch 问题,考虑一个总线结构的多处理机系统。它在总线上使用读/修改/写操作完成一次 Synch,那么每个时钟周期最多只可能完成一个 Synch 操作。若一个时钟周期是 100ns,则系统的最高性能是 10^7 SYPS(每秒同步数),即 10MSYPS。

如果一个时钟周期内多处理机的 N 台处理机能分别执行一条指令,那么整个系统的性能就执行指令数目来说是 $10N$ MIPS,但就 Synch 来说只有 10 MSYPS。该系统的 MIPS 是 MSYPS 的 N 倍。我们示例程序的一次 Synch 操作大约需要 2~3 条指令。因此,若 N 大于 3,则系统在同步方面达到饱和。反之,系统在指令的执行方面达到饱和。

合并开关提供了一种能支持并行 Synch 的机制。如果系统的时钟周期为 100ns,那么具有合并开关的系统的 MSYPS 接近达到 $10N$ MSYPS。这里的系数不一定是 10,还可能小一些。关键在于系统的 MSYPS 随 N 而增大,这就为打破 Synch 瓶颈提供了一种方法。

没有合并开关或类似能实现并行同步的机构的系统会出现如图 10.2 所示的饱和现象。这里假设系统有一个固定的 MSYPS 最大值,它与处理机台数无关。随着处理机台数的增加,系统的 MIPS 值线性地增长,但 MSYPS 值固定不变。最终,由于 MSYPS 值已达到极限值,再增加处理机的数量也不会再提高加速比了。

从图 10.2 可以看到,在处理机的数量增加到 10 台之前,加速比是线性地增长的。其中一条是理想化的分段线性曲线,它表示加速比不可能超过这个界限。另一条是在这个界限下的实际加速比的曲线。从实际加速比曲线可以看出当处理机台数增加时,由于额外开销的增加,而 MSYPS 值固定于极限值不变,这就使加速比反而下降。

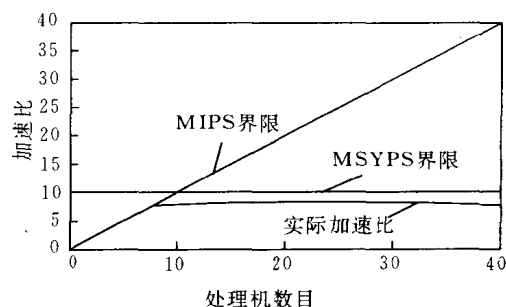


图 10.2 加速比曲线

假设程序 10.2 在一个多处理机系统上执行,其性能将受 MSYPS 瓶颈的严重限制。系统结构设计者通过什么途径来提高其性能呢? 下面三种方法是可行的:

- (1) 提高 R/C 比值,使每个同步做更多的计算。
- (2) 通过改进系统结构,以提高 MSYPS 值使系统达到平衡。
- (3) 通过降低处理机的 MIPS 值使系统达到平衡。

第一种方法最简单,并且代价小、效果好。它是通过增大粒度来改善性能的,无需增加硬件或软件的开销。这是最佳方案,下面还要详细讨论。

第二种方法是系统结构设计者在系统结构内部增加支持高 MSYPS 值的机制。合并开关是目前已经实现能使 MSYPS 值随处理机数量的增加而线性增加的唯一方法。但为满足某些特殊的应用问题,使用其它技术提高 MSYPS 值也是可能的。例如,系统结构设计者可以为同步操作专门配置一台高速的处理机,它不做别的事而专门用来管理锁和修改共享变量。系统结构设计者也可以采用一个硬件调度/分配机构来完成此任务和作处理机的管理。

第三种方法是降低处理机的 MIPS 值,这可以纠正系统的不平衡,但降低了系统的吞吐率。这个想法的根据是:如果系统的不平衡导致了有些处理机空闲,那么我们用价格便宜一点、速度低一点的处理机来替代原来的处理机,整个系统的速度仍和原来相同。本方法是要把原来低效系统变成一个吞吐率稍低一些但成本比原来低很多的高效系统。

10.1.4 增大粒度

提高程序 10.2 的 R/C 比值是很容易的。假定程序 10.2 的粒度是每一任务只有一条赋值语句,为了增大粒度,我们可以把几条语句合在一起作为一个任务,如程序 10.4 那样。

程序 10.4 采用 chunking 语句并行地解决泊松问题

```
for K:=1 to 10×M do seq
  begin
```



```

for i := 1 to M do par
  begin
    for j := 1 to M do par chunksize 50;
      begin
         $P[i,j] := (P[i,j+1] + P[i,j-1] + P[i+1,j] + P[i-1,j]) / 4;$ 
      end j loop;
    end i loop;
  end K loop;

```

程序 10.4 除最内层循环中包含一短语 chunksize 50 外,其余与程序 10.2 完全相同。这个短语告诉编译程序和操作系统把 50 个相邻的下标值合并成一组分配给一个任务,而不是把一个下标值分配给一个任务。这样,最后一个任务接收剩余的可能的少于 50 的下标值。把 chunksize 设置成 50,则程序 10.4 的 R/C 比值是程序 10.2 的 50 倍,MSYPS 请求下降为 $1/50$,当然并行性也降低为 $1/50$ 。但问题是原来并行性虽高却不能被充分利用,那么现在并行性降低了一些是可以接受的。

作为例子,我们考虑当程序 10.2 和 10.4 中的 M 为 100 时可能达到的并行性有多高。程序 10.2 的两层内循环生成 10 000 个任务。实际所生成的任务数量取决于程序本身,而与系统结构无关。如果系统中可用的处理机数量小于 10 000 的话,这是很可能的情况,那么有些任务必须入队,然后还必须出队,这就需要 10 000 次额外开销,用来对它们进行管理。程序 10.4 的程序员可以通过改变 chunksize 的大小来控制所产生的独立任务的个数和各个任务的 R/C 比值,以便降低额外开销。

假设程序 10.4 生成了 200 个任务,这对于有 200 或更多台处理机的系统来说是合适的。如果系统的处理机数量小于 200 的话,则 chunksize 的值应该设得更大一点。理想的情况是 chunksize 动态可变,它是实际可用于执行循环体的处理机台数的函数。

采用小粒度的目的是为了提高并行性,但若并行性超过可被利用的程度,那么这种提高就毫无意义了。当然粒度也不能太大以至于所生成的任务数目比可用的处理机台数还少。程序员可以选择 chunksize 的大小,也就是说程序员可以通过实验确定出在某一系统结构上处理某个应用问题的最佳粒度。

粒度仅仅是程序员所要考虑的许多因素之一。我们还没有讨论和本地存储器及全局存储器有关的问题,也还没有讨论怎样分配数据才能减少访存冲突的问题。当程序员通过选择 chunksize 来确定粒度时,他实际上把多个循环合在一起构造了一个运行环境。在这一环境下,一些数据在送回全局存储器之前,可能被重复使用过多次。在这种情况下,任务可以按如下方式组织:

- (1) 当全程变量要被修改时,要对其加锁。
- (2) 把这些全程变量从全局存储器读到本地存储器。
- (3) 进行计算,修改局部变量。
- (4) 根据局部变量的值修改全程变量。
- (5) 对加锁的全程变量解锁。

计算过程中,由于所有进程都只访问本地存储器,所以访问共享存储器的冲突很少发

生。但是加锁、同步和解锁的开销会降低系统的性能。性能究竟下降多少,在很大程度上与处理机由于等待解锁而处于空闲状态的时间有关。

如果程序员可以通过选择较大的 chunksize 来产生一个较大的任务,那么这比程序 10.4 有更大的灵活性。程序 10.4 仅仅把计算长方形一列值的循环合在一起成为一个任务,其实程序可以重新组织,使计算一个子矩阵的值的循环合在一起成为一个任务。那么任务究竟多大才算合适呢? 最佳大小的任务应该有好的粒度。某个任务内的数据操作与其它处理机的任务内的数据操作的相互干扰应该最小。互相干扰程度和系统结构及数据在存储器中的分配有关。系统结构设计者必须知道,哪些量可以供程序员选择,这样可以设计出程序员有一、两个选择使一系列应用问题都能有效处理的系统结构。

程序员有一种更有效地控制 R/C 大小的方法,即控制 chunksize 的大小和决定一个任务内哪些语句组成一组。如果系统结构的 chunksize 大小是固定的,如前面提到过的细粒度系统结构,程序员就失去了调整 R/C 比值以得到最高性能的灵活性。第一代和第二代多处理机系统应该把 R/C 比值的大小交给程序员来控制,这样才能取得足够的经验研制出最佳 R/C 比值的机器。

第二种避免 MSYPS 瓶颈的方法是减少同步的开销,或者提高系统的 MSYPS 值。这个问题非常复杂,我们将在后面对此问题进行深入的讨论和研究。

最后一种方法是通过降低和同步机制有关的处理机的速度,使得系统平衡。如果原来的 MIPS 值和 MSYPS 值不匹配的话,现在由于 MIPS 值下降了,使得系统比较平衡了。

图 10.3 表明了加速比是时钟周期的函数。我们可以看到随着处理机速度的减慢,系统的加速比反而增大。加速比是一个衡量有 N 台处理机系统的速度和只有一台处理机系统的速度的尺度。图 10.3 是 N 为 100 的情况。因为时钟周期沿 x 轴方向增大,所以图中右边的处理机的速度比左边的处理机的速度要慢。

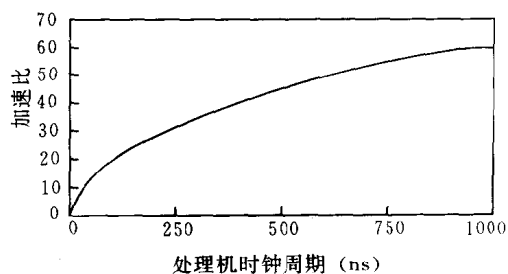


图 10.3 加速比与时钟周期的关系曲线

图 10.3 告诉我们:由 100 台慢速处理机组成的多处理机系统的加速比,要比由 100 台快速处理机组成的多处理机系统的加速比高。但加速比和性能不是一回事。虽然从 100 台慢速处理机系统所获得的加速比大于从 100 台快速处理机系统所获得的加速比,但 100 台快速处理机系统的性能比 100 台慢速处理机系统的性能要高。位于图 10.3 左边部分的系统,其快速的处理机和慢速的同步机制很不匹配,在计算过程中,处理机有许多空闲时间。再往这种系统中增加处理机并不能有效地改善性能,所以加速比很低。

当我们从图的左边部分向右边部分移动时,系统的瓶颈由同步机制转移到处理机本身。当处理机的性能成为瓶颈的主要成分时,通过增加处理机的台数可以缓解瓶颈,加速比也将提高。

从图 10.3 得到的启发是,多处理机系统的最大 MIPS 值和 MSYPS 值必须匹配,才能适应大多数应用问题。如果系统出现如图 10.3 左边的那种不平衡情况,那么与从系统

得到的性能相比,处理机就太贵了。如果系统出现如图 10.3 右边的那种不平衡情况,处理机本身即为瓶颈,那么可以通过使用快速的处理机来加快系统的速度。

10.1.5 任务的初始化.

到现在为止我们忽略了一个重要问题,即任务初始化的机制。如果程序 10.2 或 10.4 中的 do par 结构是串行地生成任务,那么串行生成任务的开销将使 R/C 比值中的 C 增大。

如果程序 10.2 的内循环任务是串行地生成的,则生成任务需要 $O(N)$ 条指令,而全部任务本身在并行执行时只需要 10 条左右指令就完成了。

如果一台处理机执行 do par 时需要执行 1000 条指令才能生成 100 个任务,那么在了这么多时间生成任务之后,各处理机只需再执行 10 条左右指令即结束了全部工作。显然,这个程序的 R/C 比值太低,以至毫无实用性。更本质的问题在于对于并行处理的任务,串行地生成任务所需的开销难以承担。

解决这个问题一个比较好的方法是在编译时就生成各个任务,这必须假定 N 的大小在编译时已经知道。这样,一个程序的全部任务都一次生成。一般来说,任务一旦生成,它们即可并行地分配给所有处理机,这就避免了串行地初始化任务所花费的时间。

另一种开销稍高一点的方法是利用任务生成二叉树动态地生成任务,生成任务的时间是 $O(\log N)$ 。为了生成程序 10.1 最内层的 do par 循环的任务,任务生成树的根结点产生两个子任务。第一个子任务负责前半任务的生成,第二个子任务负责后半任务的生成。这两子任务又分解成四个子任务,每一个负责生成四分之一任务。经过 $O(\log N)$ 步后,不需再产生子任务,所有任务本身已被生成了。

在关于多处理机系统设计的重要文献中有关任务生成问题常常被忽略。Halstead 在 1985 年提出了一种很有趣的多处理机结构,叫作 Concert。这种结构的用户可以显式地控制任务的生成。他举了一个利用众所周知的快速排序算法进行并行排序的例子。这种方法对 M 项排序的平均复杂度是 $M \log M$ 。

Halstead 算法的第一步是对 M 项进行一趟线性扫描,产生一组可并行执行的任务。算法的第二步充分地利用并行性进行排序。不管使用多少台处理机,这一算法不可能比 $O(M)$ 更快,因此加速比也就为 $O(\log M)$ 。Halstead 指出当处理机台数不多时,加速比呈线性增长,但当处理机数量接近 $\log M$ 时,加速比则固定不变了。

加速比不能超过某个值不是由于 Concert 系统结构的缺陷造成的,而是和其它许多多处理机系统一样,它要支持任务生成树。问题出在数据表示的方法上。例中待排序的数据是以单向链表形式提供给算法的,检查数据的唯一方法是沿着指针链一个一个地进行,这样做要花费的时间为 $O(M)$ 。

在这种情况下,用串行程序设计语言表示的数据和高性能并行处理结构极不相容。Halstead 详尽地描述了 Concert 系统结构的长处,但没有指出算法中采用单向链表数据结构的缺陷。低效的数据表示方法和高效的并行性开发技术发生了矛盾。

评价多处理机系统结构的关键是考察性能与关键参数的关系,即当关键参数变化时,系统性能如何变化。这里关键参数主要是指处理机数量、 R/C 比值、数据结构的选择等。

在结束简单并行性的讨论之前,让我们看一看上面所讨论的例子在多处理机系统上处理为什么会比在近邻 SIMD 系统上以及在各种向量机上处理优越得多。

如前所述,程序 10.2 在近邻 SIMD 和向量处理机上处理最理想,但实际的应用问题很少会像程序 10.2 那样简单,边界计算往往相当复杂。更一般的情况是计算区域或者不规则,或者内部有孔,或者其它结构,这都会使求解变得很复杂。

一个区域中不同类型点的计算需要不同的程序。一个纯 SIMD 系统不那么容易处理这些不同的情况而又保持高效率。最坏情况是计算某种类型点的进程运行时,要求其它进程都停止。这样,一台 SIMD 机器可能不得不串行地计算区域 A、区域 B……,这就没有利用系统结构所提供的并行性。

多处理机系统能够为各个计算区域生成不同的程序段,并能并行地执行各区域的计算。与 SIMD 结构相比,多处理机系统具有更高的并行性。

10.2 同步技术

在一个多处理机系统中,为了求解一个较大的问题,往往将该问题分解成许多子任务,并把这些子任务分配到不同的处理机上去执行。为了协调地完成这些任务,这些进程之间需要进行通信。进程之间可以通过使用共享变量来实现信息交换。共享变量每次只能被一个进程访问,因此,当两个或两个以上进程同时访问某个共享变量时,必须采取同步措施来协调并行执行的任务。

程序设计中同步大概是最困难和最容易出错的问题。其困难性在于要考虑到多台处理机可能同时发生的所有动作,但绝大多数人很难将所有可能性都考虑到。再者,同步还和处理机之间的接口部件的结构有关,所以程序员还要了解接口的硬件结构。

我们先来看看 Dijkstra 的创造性工作。当时争论的焦点是进程之间能否用像 ALGOL 60 那种普通程序设计语言的标准操作来实现同步。Dijkstra 通过一系列合理的假设,证明了这是可能的。但是他说同步是他所写过的程序中最困难的一个程序。这个程序的语句没有使用不可中断的读/修改/写操作的指令,因为 ALGOL 语言没有提供这类基本操作。但是假设程序满足可观察性的条件,即如果处理机 A 先执行 WRITE X 操作再执行 WRITE Y 操作,那么所有其它处理机都可观察到这些 WRITE 操作的执行顺序。也就是说,如果另一台处理机先执行 READ Y 操作后执行 READ X 操作,如果它读到的 Y 是新值,那么它读到的 X 不可能是旧值。如果它读到的 X 是旧值,那么它读到的 Y 也一定是旧值。这是因为 X 在 Y 之前被修改的原因。这个假设是完全合乎情理的。但如果在设计系统结构时没有考虑这个问题,那么这种假设就不一定成立。

处理机与存储器之间采用多级开关网络的多处理机系统,就可能会违背上述假设。例如有一个先执行 WRITE X,后执行 WRITE Y 的操作序列,这两个操作可能同时发送到开关网络,而它们很快按各自不同的路径去访问不同的存储模块。假设 WRITE X 遇到“热点”而被放到缓冲队列,而 WRITE Y 顺利地到达存储器并修改了 Y。这时如果另一台处理机发出一个 READ Y 和 READ X 的操作序列,并且 READ Y 毫无困难地到达存储器,获得了 Y 的新值。READ X 绕过将 WRITE X 挂起的“热点”,找到了直接到达存储器

的路径,那么 READ X 操作得到的是 X 的旧值。显然,前述的假设在此种情况下没有成立,自然 Dijkstra 的同步算法也就失败了。

今天,几乎所有的同步都是通过某种形式的读/修改/写操作来完成的,过去有些方法因为没有读/修改/写操作指令而不得不采用许多其它操作来加以弥补。相比之下,前者要比后者更有效。因此,Dijkstra 的同步方法今天已不再那么重要了。

但就 Dijkstra 算法而言,当 WRITE/WRITE 的可观察性条件不满足而导致 Dijkstra 方案失败,这一事实并不重要,重要的是这种失败的原因。因为任何多处理机算法都可能依赖这种 WRITE/WRITE 的可观察性条件,而当这一条件不满足时,算法都将失败。程序员通常不知道他们的程序代码什么时候需要这种 WRITE/WRITE 可观察性条件。有些程序串行地执行正确,在有些并行系统中运行也正确,当移植到新的环境时却可能会失败。

下面我们将讨论四种实现同步的方法,它们分别使用下列四条指令来实现同步:

- (1) 测试与设置(test-and-set)。
- (2) 增 1 和减 1(increment, decrement)。
- (3) 比较与交换(compare-and-swap)。
- (4) 取与加(fetch-and-add)。

10.2.1 使用测试与设置指令的同步技术

实现同步的第一种方法是使用测试与设置指令。这条指令完成如下操作:

```
Definition: Test-and-Set(address, bit-position);
begin
    Temp := Memory[address].bit-position;
    Memory[address].bit-position := 1;
    Condition-code := Temp.bit-position;
end definition;
```

测试与设置指令将共享数据的一个标志位置为 1,并把该位置 1 之前的值作为条件码返回。该指令有两个参数,分别是共享数据的地址和被测试位的位置。符号“A.b”表示数据 A 的第 b 位。

为了确保同步的正确执行,测试与设置指令必须是不可中断的。也就是说,该指令一旦被启动且已完成了读操作,那么在操作数被重写回去之前,不允许别的指令对它进行任何存取。如若不然,同步就会失败。

具有高速缓冲存储器的多处理机系统必须把测试与设置指令当作一条特殊的指令加以处理。因为测试与设置指令是被用来修改共享数据的,所以必须使保存在某个高速缓冲存储器中的共享数据和保存在其它高速缓冲存储器与主存中的共享数据保持一致性。一种解决办法是测试与设置指令直接访问共享存储器,这样可以避免和高速缓冲存储器打交道。同理,把共享数据的某几位置零的操作,也应按类似的方式进行。另一种解决办法是允许共享数据进入高速缓冲存储器,同时在高速缓冲存储器一致性协议中增设必要的同步措施。

下面我们讨论怎样利用测试与设置指令对共享变量进行修改。这种程序的轮廓如下：

```
Lock(shared-datum);
Update(shared-datum);
Unlock(shared-datum);
```

我们把每个共享数据或数据结构与一位叫做信号灯的联系起来。Lock 和 Unlock 语句对数据或数据结构的信号灯进行操作,而不是对数据或数据结构的内容进行操作。信号灯是一个交通指挥者,它告诉进程是否可以通过 Lock 语句。在某一时刻信号灯只允许一个进程执行 Update 程序段。如果进程 A 成功地执行了 Lock 语句,那么所有其它进程就必须在 Lock 语句处等待,直到进程 A 执行完 Unlock 语句为止。

Lock 语句可以用测试与设置指令来实现。测试与设置指令强行把共享数据的信号灯设置为 1,而不管在此之前它是否已被置 1。如果由条件码返回的设置前的信号灯值为 0,则进程可以通过 Lock 语句。

如果几个进程同时请求执行 Lock 语句,由读/修改/写操作的性质决定了这些请求必须串行地响应。在这一串请求中只有一个进程检测到信号灯为零值而通过 Lock 语句,进行修改共享变量操作。当该进程执行 Unlock 语句后将信号灯的值变为零,于是允许另一个进程通过 Lock 语句。

Lock/Unlock 语句对各自至少占一条指令。根据修改的情况不同,被 Lock/Unlock 语句保护起来的修改程序段可能是 2~3 条指令,也可能是 10~100 条指令。那么这个串行程序段就需 5~100 条指令,或更多的指令。

在一秒钟内顺序地执行的串行程序段的数目称为 MSYPS。它是处理机的 MIPS 慢 1/5~1/100。对于单处理机系统来说,如果 MSYPS 很高,比如说是 MIPS 的 10%,那么 MIPS 很可能成为系统的瓶颈。如果 MSYPS 较低,比如说是 MIPS 的 1%,那么 MSYPS 很可能成为系统的瓶颈。

在多处理机系统中,MIPS 的峰值大小与应用问题所用的处理机数量成正比关系。绝大多数系统的 MSYPS 有一个固定的极限值,它和应用问题所用的处理机数量无关。如果我们把注意力集中于 MIPS,而忽略了 MSYPS,那么我们可能认为一个程序分配到的处理机越多,获得的机器解题能力越强,但实际情况不这样。

分配到的处理机越多,程序可用的内存空间越大,MIPS 也越高,但 MSYPS 值可能一点也没有提高。如果 MSYPS 是系统的瓶颈,那么新增加的处理机并不会使计算速度加快。事实上,由于处理机之间的竞争、Lock/Unlock 和对共享数据修改操作等反倒会占用更多的时间,反而使总的运算时间增加而不是缩短。

MSYPS 瓶颈仅仅是导致性能下降的几个潜在根源中的一个。例如,当一台处理机被 Lock 阻塞时会发生什么情况? 该处理机可能会被用来做其它有用的工作,虽然 MSYPS 的瓶颈仍然存在,但系统的 MIPS 还是增加了。测试与设置指令只完成有关 Lock 的一半操作,另一半操作要视 Lock 是否被允许通过而定。如果测试与设置指令返回的信号灯的原始值为零,则 Lock 允许通过,处理机执行修改段程序。如果 Lock 不允许通过,至少有两种不同的方案可以采取:

(1) 自旋锁(spin lock)方案。它是重复测试 Lock,直到 Lock 被允许通过为止。

(2) 任务入队(enqueue a task)方案。它把被阻塞的进程挂起,将它的状态放到一个与信号灯有关的队列中去。把处理机重新分配给当前任务队列中等待执行的任务。

自旋锁方案很浪费计算周期,还会引起对信号灯存储单元的竞争。当多台处理机等待一个信号灯时,这种竞争使得正要企图释放 Lock 的进程也被延迟了,这会降低 MSYPS,并使信号灯的瓶颈更严重。

任务入队方案看起来似乎比较高效,因为它把可用的周期用来做其它有用的工作。但实际上入队/出队的消耗很大,甚至比自旋锁方案所浪费的周期还要多。更为糟糕的是为了将任务入队,处理机必须存取和修改共享的队列指针,而这种访问本身还包含某种 Lock/Unlock 操作。如果这个 Lock 请求也未被允许的话,我们就面临着把任务入队这件工作本身也当作任务进入队列的问题。这可能会无限循环下去。显然,在某一层次,比如说在第一层或第二层,我们就不得不采用自旋锁方法而不是任务入队方法。

从性能角度来看,上述两种方法对 MIPS 和 MSYPS 有截然不同的影响。任务入队方法把空闲的处理机重新分配去做其它有用的工作,所以会提高 MIPS。而自旋锁方法因为重复地测试信号灯而浪费了有用的机器周期,所以降低了 MIPS。这种截然不同的作用还体现在对 MSYPS 的影响上。任务入队方法使得被 Lock 保护起来的临界区的个数和长度都增加了。临界区个数的增加,使得 MSYPS 的需求增加了。由于一台处理机在某一时刻只能执行一个临界区程序,由于入队和出队过程需要完成许多种操作,这使得临界区程序段长度增加,最终使 MSYPS 的最大值降低了。

如果一个并行进程主要是受到 MSYPS 的限制,而不是 MIPS 的限制,那么把自旋锁方法改为任务入队/出队方法将导致吞吐率下降。相反,如果限制来自 MIPS 而不是 MSYPS,那么把自旋锁方法改为任务入队/出队方法将导致相反的效果。如果入队/出队的消耗很低,以致于一个系统即使采用任务入队/出队方法,也仍然主要受到 MIPS 而不是 MSYPS 的限制,那么这种改变可能会使性能提高。

下面我们简单地讨论 Unlock 的实现问题。因为与之相对应的 Lock 有自旋锁和任务入队锁之分,所以 Unlock 的实现也将会有很大的不同。如果要解开一个自旋锁,那么上锁的处理机只需把信号灯置为零,而没有必要通过 READ/MODIFY/WRITE 操作来解锁。当 N 个进程都要循环测试一个信号灯时,这将导致性能下降,即解锁进程会和这 N 个都要访问信号灯的进程发生冲突,使得解锁进程延迟了,延迟的时间与 N 成正比。为了避免这个问题,系统结构设计者可以给 WRITE 请求较之读/修改/写操作更高的使用存储器的优先权,当然其它仲裁规则要保证每个请求最终都可以得到服务。

如果 Lock 操作把挂起的任务入队,那么 Unlock 操作就应使一个等待那个信号灯的任務出队。Unlock 操作要检查共享队列指针,所以任务出队需要读/修改/写操作而不是简单的写操作。

10.2.2 使用增 1 和减 1 指令的同步技术

实现同步的第二种方法是使用增 1 和减 1 指令。它比测试与设置指令有更强的功能。实现这两条指令的具体方法是每条指令在执行期间“拥有”一个指定的存储单元。当

指定的存储单元正在被 READ 访问时,其它任何指令都不能访问它,直到修改过的内容重新写入该单元为止。

普通的增 1 和减 1 指令只能简单地修改操作数,而不需具有读/修改/写的不可中断性质。这种普通的增 1 和减 1 指令在多处理机系统中可以自由地用于修改非共享数据,而不必考虑其用法的正确性。

仅当指令具有不可中断性质,它才能用于修改共享数据。但即使一条不可中断指令的具体实现不正确,或者程序员把一条可中断指令误认为是不可中断指令,该指令几乎总能正常工作。但在极少数情况下,从另一台处理机发来的访问请求恰好发生在增 1 或减 1 指令的 READ 和 WRITE 操作之间,那么程序将会出错。一个完美无缺的程序必须绝对无错,这就要求用不可中断读/修改/写指令来实现同步。

在具有不可中断的增 1 和减 1 指令的计算机系统中,用增 1 和减 1 指令来实现同步,要比用测试与设置指令来实现同步所需要的指令要少。因为测试与设置指令返回的信息只有一位,而增 1 和减 1 指令返回一个存储单元的全部内容。因为返回的信息多,所以实现同步所需的指令条数少。

例如,有一个长度为 M 的共享缓冲区,如果每个进程是对不同的单元进行操作,则 M 个进程可以同时进入该共享缓冲区。如果在该缓冲区中已经有 M 个进程在活动,若又有一个或多个进程请求访问时,第 $M+1$ 个进程和以后的进程必须等待。从本质上说,为了实现上述功能,我们需要一个扩充的信号灯。

用测试与设置指令实现的信号灯只允许一个进程通过,在信号灯被解锁之前,禁止其它进程访问。这种信号灯只适宜于控制一个长度为 1 的缓冲区。扩充的信号灯允许 M 个进程同时通过。如果已经有 M 个进程正在长度为 M 的共享缓冲区活动,那么在信号灯被解锁之前,以后的进程被禁止通过。如此每解锁一次允许一个进程通过该信号灯。

有一种使用增 1 和减 1 指令实现这种扩充信号灯的简单方法。开始时给该信号灯赋一个初值 M ,然后每个请求进程对信号灯进行减 1 操作。如果在减 1 操作之后,发现信号灯是一个非负的数,则该进程可以访问该缓冲区;如果信号灯是一个负数,则该进程被阻止访问。被阻止访问的进程随后立即对信号灯增 1,以表示该进程没有对缓冲区操作。如同我们早先讨论的测试与设置指令一样,被阻塞的进程可以入队或不断测试信号灯。一台处理机在完成对缓冲区的访问之后,对信号灯增 1,以表示有空间可供其它进程使用。

这种简单的同步实现方法有时会出现一种称之为活锁(live lock)的错误。下面我们以程序 10.5(a)为例来讨论。在程序 10.5(a)中,当进程要访问缓冲区时,对信号灯进行减 1 操作。如果减 1 后信号灯的值是负数,则对信号灯进行增 1 操作,然后重复测试信号灯。如果减 1 后信号灯的值是非负数,处理机便可进入被保护的程序段。退出时对信号灯进行增 1 操作。

问题是系统可能进入一种状态:一方面缓冲区中有可用空间,另一方面有用工作无法进入缓冲区,这就是“活锁”。这里的活锁是相对于死锁状态而言的。如果存在一个循环: A 在等 B, B 在等 C, ..., 周期的最后一项在等 A,这就产生了死锁。死锁是死的,因为这种状态是永久的。在死锁循环中,除非有一个或多个进程放弃要求,否则所有进程都无法结束这种死锁状态。活锁则不一样,它不是固有的,系统进入活锁状态是因为遇到一种特殊

的时序。

下面我们分析程序 10.5(a) 进入活锁状态的原因。我们假设在信号灯为零时有大量处理机同时对该信号灯申请减 1 操作, 那么信号灯值将变为 $-\infty$, 它还能变为正数吗? 如果每台被阻塞的处理机先执行增 1 操作, 然后返回去重新测试信号灯并执行减 1 操作, 这整个过程是不可中断的, 然后才把信号灯传送给下一台处理机, 那么信号灯值将从 $-\infty$ 变为 $-\infty + 1$, 然后又回到 $-\infty$ 。如果这时有 M 台处理机同时从缓冲区退出, 它们将信号灯值增加到 $-\infty + M$, 但这仍是一个负数, 所以不允许处理机访问缓冲区。这时就出现有用的工作因为事件发生的顺序不合理而被阻塞的现象。如果改变事件的顺序, 就可以使信号灯非负, 进而使有用的工作只要缓冲区有空间就能开始执行。

程序 10.5 有活锁现象的同步和无活锁现象的同步

```
while decrement(semaphore) < 0
do increment(semaphore);
{临界程序区}
increment(semaphore);
(a) 有活锁现象的同步

Loop: while semaphore > 0 do;
if decrement(semaphore) < 0 then
begin
increment(semaphore);
go to Loop;
end;
{临界程序区}
increment(semaphore);
(b) 无活锁现象的同步
```

程序 10.5 中指令 increment 和 decrement 是不可中断的读/修改/写指令。参数 semaphore 是信号灯变量。

程序 10.5(b) 采用了一种能消去 10.5(a) 活锁现象的机制。它是在信号灯减 1 操作之前测试信号灯。最坏的情况是大量的处理机发现信号灯为非负时, 对信号灯进行减 1 操作而使它变为 $-\infty$ 。一旦所有处理机完成信号灯减 1 和增 1 操作后, 准备对它进行重新测试时, 发现信号灯为负值, 任何减 1 操作不再允许。这样, 当该值变为非负时, 至少有一个进程可以在该值再次变为负数之前获准通过。因此有用的工作可以继续执行, 不会出现活锁现象。当然, 在最坏情况下, 处于“活状态”的处理机的平均台数会低于最大值。

10.2.3 使用比较与交换指令的同步技术

对一台传统的处理机来说, 一条比较与交换 (compare-and-swap) 指令可将一个被锁的程序段压缩成单条指令——比较与交换指令, 因此该指令将产生最大的 MSYPS。共享数据在比较与交换指令的开始时被加锁, 在指令执行期间被修改, 在指令结束时被解锁。

这与早先介绍的同步技术不同,它们是在共享数据的修改前后,通过控制信号灯建立临界区。在有些应用场合,如任务的入队/出队等,比较与交换指令非常有用。

程序 10.6 定义了比较与交换指令的操作。由此定义可知该指令需要两个寄存器:一个用于保存共享数据的原值,也称老值;另一个用于保存新值。

用比较与交换指令来修改共享数据的目的是无需对共享数据加锁就可以利用普通指令来计算共享数据,然后用不可中断的比较与交换指令重新取出共享数据。检查该数据是否被其它进程修改过,如未被修改过,则执行修改操作。如被修改过,则把新取来的共享数据装入保存原值的寄存器,然后重新计算一次新值,再用一条比较与交换指令来完成修改。

程序 10.6 比较与交换指令

```
Definition: Compare-and-Swap(Address,Reg-old-val,Reg-new-val);
temp := Memory[Address];
if temp := Reg-old-val then
  begin
    Memory[Address] := Reg-new-val;
    Condition-Code := 1;
  end
else
  begin
    Reg-old-val := temp;
    Condition-Code := 0;
  end;
end of definition;
```

在程序 10.6 中,变量 Address 是存储器地址。Reg-old-val 和 Reg-new-val 是机器的两个寄存器。比较与交换指令一旦启动,就不可中断。该指令结束时通过对条件码的测试可以确定共享数据的修改是否发生过。

程序 10.7 是使用比较与交换指令的一个简单例子。这个程序把一个局部计算出的增量加到共享变量上去。程序首先把共享变量的当前值读到 Reg-old-val 寄存器,然后把局部增量与 Reg-new-val 寄存器内容相加,把相加的结果存入 Reg-new-val 寄存器,最后利用比较与交换指令修改共享变量。如果比较与交换指令的返回值为 1,表示共享数据没有被其它进程修改过,那么本次修改成功了,否则程序返回到 Loop,重新进行计算。从程序 10.6 我们知道比较与交换指令本身会把共享变量的当前值加载到 Reg-old-val 寄存器,所以计算时没有必要再次读共享变量了。

程序 10.7 用比较与交换指令修改共享累加和变量

```
Local-sum := 0;
for i := 1 to N do
  Local-sum := Local-sum + x[i];
  Reg-old-val := Memory[Address];
```

```

Loop: Reg-new-val := Local-sum + Reg-old-val;
Compare-and-Swap(Address, Reg-old-val, Reg-new-val);
if Condition-Code = 0 then go to Loop;

```

程序 10.7 中的 Address 是全程累加的存储器地址。

下面把程序 10.7 修改共享变量的方式与早先通过 Lock, READ, MODIFY, WRITE, UNLOCK 等操作来修改共享变量的方式进行一下比较。使用 LOCK/UNLOCK 对时,在一个时刻只有一台处理机能够执行包含 READ, MODIFY, WRITE 操作的指令。使用比较与交换指令方式时,许多台处理机可以并发地执行程序 10.7。由于许多处理机都可以对共享累加和进行读和写,所以有可能出现这种情况,即在程序 10.7 的开头某一时刻,一台处理机读了共享累加和值,执行比较与交换指令时对共享累加和进行修改,在这期间另外一些处理机可能修改了共享累加和。这里我们没有用 LOCK 来禁止上述这种并行访问。

保证程序正确性的关键是利用比较与交换指令进行测试。因为共享变量的新值是共享变量的老值和一个数相加的结果,所以要保证老值没有被其它处理机修改过,这由比较与交换指令通过测试来发现。如果没有被修改过,则计算所得的新值是正确的,把新值存入共享变量就可以了。

比较与交换指令的最有意义的应用是不需要加锁和解锁操作即可实现入队和出队。因为队列指针是共享变量,所以典型的入队/出队程序在修改队列指针前要加锁。这种方法因为限制了计算机系统的最大 MSYPS 值,而使队列程序成为多处理机系统的瓶颈。在程序 10.7 中,我们把修改累加和的程序段压缩成为一条比较与交换指令。与此类似,也可以用比较与交换指令的方法并发地修改队列指针。

程序 10.7 看起来很简单,但要正确地使用比较与交换指令要有一定的技巧。其难点在于该程序存在各种各样并发执行的可能性。总之,比较与交换指令的思想是要提高并行性,但当许多处理机并发地执行同一个程序时,许多事件可能以程序员不可预知的顺序发生,因而有可能导致同步失败。可以说比较与交换指令是多处理机系统最有效的软件工具,也是最难使用的工具。

为了弄清楚使用比较与交换指令的有效性和危险性,我们以数据入队列问题为例进行说明。图 10.4 显示了队列的数据结构并说明了使用比较与交换指令高并发地完成数据入队列的过程。图 10.4(a)显示的是一个单链的队列,头指针 Head 指向队列的第一项,这一项是待移走的项。尾指针 Tail 指向队列的最后一项,如果有新的数据要入队就加在尾指针指向的单元。

把一个数据项插入队列只要执行下面三条语句就可实现,即把 Item 单元的内容加到队列的队尾。

```

Memory[item].Link := nil
Memory[Tail].Link := item;
Tail := item;

```

符号.Link 是一个指针域,其值为后继数据项的地址。后两个语句因为 Tail 是可读、

修改、写的共享变量,所以执行时不可中断。

当程序正确地执行时,插入一项数据后的结果如图 10.4(b)所示。如果处理机 1 要把 ITEM1 插入队列,处理机 2 要把 ITEM2 插入队列,假设处理机 1 先执行第二条语句读取 Tail 的当前值,接着处理机 2 执行第二条语句读取 Tail 的当前值。处理机 1 和处理机 2 仍按这个顺序执行第三条语句对 Tail 值进行修改,那么 ITEM1 将丢失。如果处理机 2 先执行最后一条语句,然后才是处理机 1 执行,那么 Tail 将指向 ITEM1,但与前面的数据项断开了。这样,所有以后的数据项将不可能再从头指针开始搜索到。这种情况如图 10.4(c)所示。其中处理机 1 插入 ITEM1,处理机 2 插入 ITEM2。两台处理机交叉地执行语句的情况。

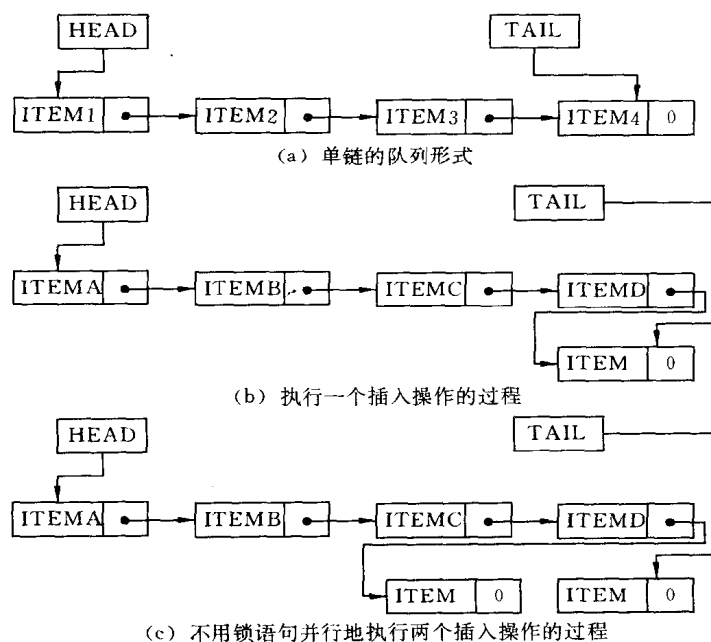


图 10.4 队列

传统的程序设计技术总是在执行前对这一段语句加锁,在执行以后对这些语句解锁。一个基于比较与交换指令的方法如程序 10.8 所示。假定正在并发执行的处理机中有一台执行比较与交换指令成功,则它读到 Tail 的值并把它的指针指向 ITEM。这使 Tail 指向新的 ITEM。保存在寄存器 Reg-Tail 中的原 Tail 仍指向原来的队尾,把这个原来的队尾与新的 item 连接起来就完成插入操作了。如果比较与交换指令失败,那么处理机用已被比较与交换指令取到 Reg-Tail 寄存器中的 Tail 新值重新计算一次。比较与交换指令的最后效果就是保证存入 Tail 中的值与 Memory[Tail].link 中的值一致。

程序 10.8 能正确地实现并发的入队(ENQUEUE)操作,但该程序没有考虑空表情况。随着程序复杂性的增加,正确地使用比较与交换指令也越来越困难了。

现在我们看一下程序 10.8 执行到比较与交换指令时的情况。如果 Reg-Tail = Tail,则表示自 Tail 上一次从存储器读出以来,还没有其它并发的 ENQUEUE 修改过 Tail 的

值。如果我们允许出队(DEQUEUE)可以和 ENQUEUE 一样并发执行,那么上述结论就不一定成立了。例如,一个 DEQUEUE 操作可能已把 Reg-Tail 中的数据项从队列中移走,紧跟着一个 ENQUEUE 操作执行比较与交换指令又把移走的数据项存入队列。这时 Tail 的值仍为先前的值,即与 Reg-Tail 的内容相同。于是,我们遇到两台处理机同时用不同的地址修改 Memory[Tail].link 的情况。

程序 10.8 用比较与交换指令实现 ENQUEUE 操作

```
Memory[item].Link := nil;
{把插入队尾的数据项初始化}
Reg-Tail := Tail;
{读 Tail 到 Reg-Tail 寄存器}
Loop: Compare-and-Swap(Tail,Reg-Tail,item);
if Condition-Code = 0 then go to Loop;
{比较与交换指令执行失败时返回 Loop}
Memory[Reg-Tail].Link := item;
```

程序 10.8 能实现并发的 ENQUEUE 操作。如果 ENQUEUE 操作和 DEQUEUE 操作并发执行,程序 10.8 有可能出错,下面我们要分析这种出错的原因。

如果程序 10.8 刚好在比较与交换指令执行之前被中断,接着有两台处理机分别执行 DEQUEUE 操作和 ENQUEUE 操作,而且 ENQUEUE 操作是把由 DEQUEUE 操作刚出队的数据项放回队列,这时候会出现上述这种错误。虽然这种情况很少发生,但仍是可能的,甚至不会被比较与交换指令检测出。

出错的原因是比较与交换指令并不具备能感知 Tail 变化历史的功能,所以一看到 Tail=Reg-Tail 就认为 Tail 从上次读出以来一直未被修改过。这一结论有时不正确。任何使 Tail 保持原状态的一系列事件都可能导致程序 10.8 失败。

一种改进比较与交换指令安全性的很实用的方法是把比较与交换指令扩充,使它能同时处理两个变量。这两个变量必须相邻存放,以便通过一次 READ 和 WRITE 操作就能完成这两个变量的存取。

程序 10.9 说明这种扩充的指令是如何改进程序的可靠性的。变量 Tail 和 Count 相邻存放。Tail/Count 对的当前值读到 Reg-Tail 和 Reg-Count 寄存器。仅在双比较与交换(Double Compare-and-Swap)指令执行之前,把 Reg-Count 的增 1 内容传送到 New-Count 寄存器。Double Compare-and-Swap 指令验证 Tail/Count 未被修改后,用 ITEM/New-Count 的内容修改 Tail/Count 的值。

因为 Double Compare-and-Swap 指令执行成功后,既修改 Tail 值又修改 Count 值,所以如果发现 Tail 和 Count 值被修改过,那么就表示其它队列操作已经发生过或正在执行。这将强行使 Double Compare-and-Swap 指令失败,转到 Loop 执行,阻止错误的修改操作。只有当 Tail 和 Count 都没有修改过才能进行修改操作。

程序 10.9 用双比较与交换指令实现 ENQUEUE 操作

```
Memory[Item].Link := nil;
{把插入队尾的数据项初始化}
```

```

Reg-Tail&Reg-Count := Tail&Count;
{把双变量 Tail 和 Count 值读到 Reg-Tail 和 Reg-Count 寄存器}
Loop: New-Count := Reg-Count + 1;
Double Compare-and-swap(Tail & Count, Reg-Tail & Reg-Count, Item & New-Count);
if Condition-Code=0 then go to Loop;
Memory[Reg-Tail].Link := item;

```

程序 10.9 中符号 Tail & Count 表示相邻存储的两个变量或两个相邻的寄存器,一次操作就能存取这两个变量。Double Compare-and Swap 指令从 Memory[Tail]中读取一个双倍长度的操作数,把它与寄存器 Reg-Tail 和 Reg-Count 中的内容比较。如果相等,则用 Item 和 New-Count 的值修改 Tail 中的内容。如果不等,则用 Tail & Count 的值修改寄存器 Reg-Tail 和 Reg-Count 的内容。如果 Count 增 1 许多次,使得 Count 产生溢出后又回到原始值,而 Tail 保持不变,当出现这种情况时,程序 10.9 将失败。System/370 的 Count 经过 40 亿次操作后才回到它的原始值,因此程序 10.9 失败的可能性非常小。当一个进程在 Double Compare-and-Swap 处被挂起,而其它处理机把 Count 增 1 操作 40 亿次,而且 Tail 保持不变,此时程序 10.9 才出错。

现在总结一下用比较与交换指令实现同步的特点。首先,使用这条指令非常有效。但同时又非常危险,容易出现很难检查出来的错误。所以只有那些很有经验的程序设计者且要非常仔细才能使用它。

10.2.4 使用取与加指令的同步技术

以上所讨论的三种同步技术都有一个共同的特点,那就是它们都属于串行方法,即在一个时刻,只有一台处理机在执行读/修改/写操作。

取与加操作与上述三种方法不同,它可以真正并行地执行。假设多台处理机要修改同一个变量,那么它们可以同时执行取与加指令。但是,每条取与加指令的操作数不是同一个变量并且这些变量存放在同一个存储器中或需共享访问通路,那么对这些变量进行取与加操作也必须串行地完成。

取与加指令的格式如下:Fetch-and-Add(Sum, Increment),它是把增量值 Increment 加到共享变量 Sum 上。如前面所述,这一加法操作能并行地进行,既不需要加锁和解锁,也不像比较与交换指令那样有时需要重复测试和循环。

就性能而言,如果某一时刻仅有一台处理机要求修改 Sum,那么用比较与交换指令要比用取与加指令有效。这是因为这种情况下硬件网络所产生的延迟时间不大。但是,同时存在 10~100 个访问请求时,取与加指令将比较与交换指令速度快,因为前者能同时处理所有请求。

对一个处理机数目较少的系统来说,采用比较与交换指令可能是最佳方案。但随着处理机数目的增加,采用取与加指令能改进系统的性能。处理机台数越多,采用取与加指令也越有吸引力。但是采用取与加指令的性能价格比仍是一个值得研究的问题。它的实现代价很高,而且只有当多台处理机同时访问同一个共享变量时才能发挥它的作用。如果多台处理机访问同一存储器中的不同变量,那么取与加指令一点也派不上用场。

如果处理机数目 N 很大,例如 $1000 \sim 10000$,那么确实需要采用取与加指令或其它等价的并行同步机构。如果没有这一类机构,对于 1000 台处理机的系统,有限的 MSYPS 将严重影响系统的性能。

为了进一步弄清取与加指令的工作方式,我们再次考虑共享队列的入队和出队问题。采用比较与交换指令方法是面向指针的,它把链当作地址处理。取与加指令更适合于计数器而不是指针,这里的计数器是一个由加减控制的变量。如果要求一系列计数操作的最后结果不受加减运算的顺序的影响,必须使用取与加指令。

我们希望提出一些算法,使得任意顺序的运算其结果都是正确的。用取与加指令实现入队的最佳方法是以计数器为基础。这种实现方法的基本思想是使用一个计数器 Tail,执行入队操作时,计数器 Tail 增 1。Tail 的值是一个偏移量,用来表示下一个项的插入位置。下面是用取与加指令实现入队操作的简单但不完全的程序。

```
Procedure Enqueue(item,Queue);
begin Place := Fetch-and-Add(Tail,1);
  Queue[Place] := item;
end of Enqueue;
```

取与加指令使 Tail 增 1,同时返回增 1 以前的 Tail 的值。该返回值是偏移量用来表示插入点的位置。如果 N 台处理机同时执行取与加指令,Tail 接收到的是增量的和,而返回给每一台处理机的是不同的位置值。这样,每台处理机在队列中有不同的插入位置。

这是用取与加指令实现入队操作的最基本思想。但是,完整的实现因为需要考虑各种不同的条件而变得非常复杂。这些条件包括:

- (1) 队列应该是循环的,这样当 Tail 值超过 Queue 向量的长度时,把 Tail 值置为 0;
- (2) 队列中活动项总数不能超过 Queue 向量的长度;
- (3) Dequeue 操作允许把多个项从队列中并行地移出;
- (4) 不允许对一个空队进行 Dequeue 操作;
- (5) Enqueue 和 Dequeue 操作都必须避免出现活锁。

如果用变量 Tail 和 Head 分别控制队列的插入位置和删除位置,那么队列中现有项的数目正好是 Tail 和 Head 的差。因为当 Tail 和 Head 的值超过队列的长度时,要重置为 0,所以两者的差实际上等于活动事件数按队列长度取模后的值,所以从 Head 和 Tail 值判别出活动事件的数目需要一定的技巧。如果用一个变量 Count 来表示活动事件的数目就非常容易了。入队和出队操作实际上是用取与加指令对 Count 变量分别进行增 1 和减 1 运算。由取与加指令返回的值可用来控制队列的上溢和下溢。

为了避免出现活锁现象,入队操作必须在对 Count 增 1 之前对它进行测试。出队操作必须在对 Count 减 1 之前对它进行测试。当队满和队空时,程序应返回去测试 Count 值,这与用增 1 和减 1 指令处理活锁现象类似。同时,当队满和队空时,其它处理机应返回去重新执行其操作。在这种方式下,处理机将在最外层不断进行测试,在 Count 达到安全值以前不会对它进行增 1 和减 1 操作。

为了处理队列的循环问题,当一条取与加指令增 1 Head 而使它超过队尾时,正在并

发地访问 Head 的一组处理机将发现 Head 的值小于、等于或大于队列的长度。得到合法 Head 值的处理机将继续执行下去,那些发现 Head 值超过队尾的处理机通过减 1 Head 值来放弃当前的活动,并返回到程序的以前位置,在队列中申请一个位置。最终,Head 将返回一个最小的不合法值。

实现入队/出队的完整算法由以下几部分组成:

- (1) 管理 Count, Head 和 Tail;
- (2) 处理队列循环、队列空和队列满;
- (3) 避免出现活锁。

我们曾说过正确地使用比较与交换指令非常困难,但是与使用取与加指令的难度比起来要小得多。在比较与交换指令的执行前后很容易因并发性出现错误。由于比较与交换指令必须串行地执行,所以还较容易验证比较与交换指令的算法正确性。取与加指令支持的并行性较比较与交换指令支持的并行性会高得多。

许多处理机可能对同一个数据并发地执行取与加指令,这使得要考虑的各种情况大大增加,从而使验证工作更加困难。即使熟练的程序设计者也必须非常谨慎地使用取与加指令。因为大多数程序设计者不可能直接使用取与加指令设计出正确有效的程序,所以采用取与加指令的同步,大都采用库调用方式来实现。

10.3 并行搜索

并行处理机最明显的一个用途是搜索。许多研究报告表明,并行处理机解决搜索问题时有很高的计算速度,其主要依据是搜索过程中处于忙状态的处理机的个数很多。然而遗憾的是多处理机系统的加速比与处于忙状态的处理机数目并不是一回事。这是因为有些处理机虽处于忙状态,但它们所做的工作可能毫无用处。

这一节将介绍两个不同的搜索算法。第一个算法是搜索一个函数的极大值,从这个问题我们会惊奇地发现即使最佳搜索算法,也只能产生 $O(\lg N)$ 的加速比。更令人惊奇的是所有处理机在算法的每一步都处于忙状态,因而计算的巨大浪费被掩盖了。第二个算法是一个很成熟的搜索算法,这里用它来说明应该如何开发有用的并行性问题。

10.3.1 搜索单峰函数的极大值

Karp 和 Miranker 在 1968 年曾研究过用 N 台处理机搜索单峰函数的极大值问题。图 10.5 是典型的单峰函数。根据定义,一个单峰函数或者单调或者在其两端点之间存在一个极大值点。我们的目标就是找到极大值点的 X 坐标。搜索工作在多处理机上进行,其中每台处理机都能计算区间内任意一点 x 的函数值 $f(x)$ 。假设求值所需要的时间是固定的常数,这样所有处理机的计算工作同时开始和同时结束。在求出一个值以后,处理机之间要互相交换信息以决定下一个计算点。我们还假定交换信息所花时间也是一个常数。

整个搜索算法就是各台处理机计算和交换信息过程的不断循环,直到找到极大值为止。Karp 和 Miranker 的研究结果表明,最佳策略依赖于处理机台数的奇偶性。但不管处理机台数是奇是偶,最佳策略的加速比最大只能达到 $O(\lg N)$ 。

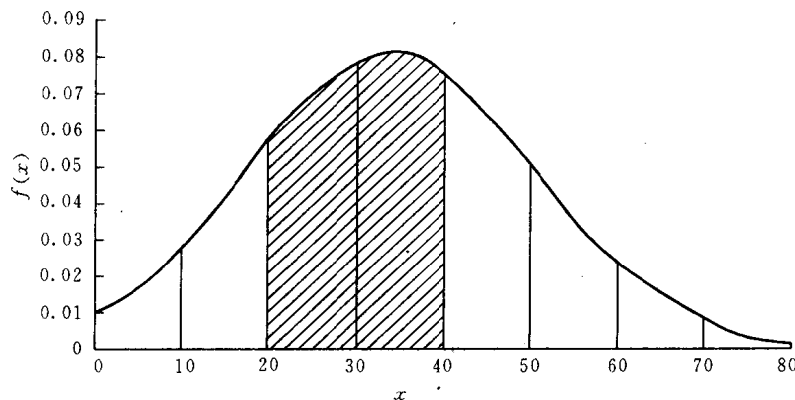


图 10.5 搜索单峰函数的极大值

从表面上看,在上述算法的每一步,所有处理机都处于忙状态,令人难以相信加速比只有 $O(\lg N)$ 。实际上,如果使用足够多的处理机,我们一步就能找到极大值点,这是最快速度了。但如果单台处理机采用二分法搜索,在 $O(\lg N)$ 步之内也能找到极大值点,显然,加速比最大只能达到 $O(\lg N)$ 。

图 10.5 说明了这种算法的执行情况。图中 7 条垂线表示 7 个同时开始搜索的点,这些点是均匀分布的。其实这些点根据情况可以不均匀分布。

返回信息的性质非常重要。从这一组搜索点可以肯定极大值一定在阴影部分的某一位置上。其理由是:我们可以通过检查两个相邻的函数值就可求出函数的导数值。在函数的极大值处,导数值等于零。只有在阴影区域内,单峰函数的导数值可能是零。因此算法的下一步就是 7 台处理机去计算阴影部分的函数值,如此重复这一过程。

从上述过程我们可以看到有些工作是浪费的。真正对指导搜索有用的信息是导数值改变符号的那部分。这样,在导数值变号处附近的一些处理机工作是有用的,而远离导数值变号处的那些处理机所做的工作毫无用处。

我们已知函数是单峰函数,所以如果函数在 x 点的导数为负,则在所有 $y > x$ 的点的导数值都为负。因此,所有在 x 点右边的处理机所做的都是无用功。同理,如果函数在某点 x 处的导数值为正,则在所有 $y < x$ 的点的导数值都为正,那么所有在 x 点左边的处理机所做的都是无用功。

让我们从可利用和真正有用的信息角度来考察这一问题。 N 台处理机中每台只返回最重要的一位信息,即函数的导数值的符号,因此并行算法的一步可以计算 N 位信息。然而这些信息并非相互独立的。实际上, N 台处理机将 X 轴分为 $N+1$ 个区间,下一步即在这 $N+1$ 个区间中选择相邻的两个区间作为搜索对象。这样一共存在着 N 种选择。这 N 种选择的信息量是 $\lg N$ 位,而不是 N 位。因此,我们费尽心机得到 N 位信息只有 $\lg N$ 位有用,也就是说,此算法在每次迭代过程中都舍掉 $N - \lg N$ 位的信息,此算法有一定的浪费。

我们研究的是单峰函数,获得的信息有冗余。但是假如考虑的是一个多峰函数,要求出整个范围内全部的极大值,那么每台处理机的工作就不再是冗余了。

尽管如此,单峰函数仍然十分重要。在数据库中按分类码查找的就属于这种函数。当分类码与数据库码相比较时,要计算差值,下一步查找与这个差值的符号有关。差值的绝对值是单峰函数。在这种情况下,单峰函数有一个极小值而不是极大值。

Krap 和 Miranker 的研究结果表明,多处理机在对一个按单一查找码排序的文件进行多路查找时并不十分有效。相反,多处理机应该用来进行多个相互独立的查找。因此,不能期望多处理机完成单一查找码的查找比一台单处理机完成得更快,但多处理机可以高效地实现并行查找。

如果数据库是按某种码排序的,查找码和数据库码之间的距离是单峰函数,这正好和 Karp-Miranker 的模式相符。如果数据库未经排序,那么得到的是一个多峰函数,Karp-Miranker 的假设不成立,串行查找就必须查遍整个数据库。在这种情况下,多处理机具有巨大的潜力,有很高的加速比。

因此,尽量采用多处理机对未经排序过的数据库进行查找以获得高性能。然而,在这种情况下,从并行查找所得到的并不是所看到的那种加速比,而是节省了对数据库排序和维护这种次序所需要的开销。如果这种开销很小,则并行算法的收益就小,如果开销很大,则并行算法的收益就很大。

在这里观察到的很重要的一点是,并行技术仅仅是许多解决问题的技术中的一种,和好的串行技术相比,它亦可能相当糟糕。评价并行算法的标准是性能/价格比,而不是性能本身。所有处理机的所有算法都可以用这一通用标准来评价。例如当我们得知用 1024 台处理机并行查找比串行查找要快得多,假如可以获得 10 倍的加速比时我们会很感兴趣,但假如我们得知仅用 32 台处理机就可以获得 5 倍的加速比的时候,对前者就不感兴趣了。

10.3.2 并行分支限界法

我们以旅游商问题为例进行讨论。旅游商问题是图论中的一个数学问题。该问题为:给出 N 个城市和每一对城市之间的距离,一个售货员从城市 c 出发,访问每一个城市且仅访问一次,最后再回到出发城市 c ,求整个旅行的距离为最短的一条路线。旅游商问题容易使人误解,因为它概念简单而且易于描述,但却很难解决。众所周知,旅游商问题属于 NP 完全类难题。解决这种问题即使用最好的算法,在最坏情况下,计算时间随着输入量的增加而增加,增加速度比任何多项式函数的增加还要快得多。许多学者深信 NP 完全类问题的计算时间实际上随着输入量的增加至少按指数函数关系增加。

下面将要描述的算法是一个很好的算法,它的平均复杂度只有 $O(N^3 \lg N)$,比问题大小的平方还要小,问题本身的大小为 $O(N^2)$ 。这似乎与问题的难度为 NP 相矛盾,但实际上并非如此。虽然此算法的平均复杂度很低,但最坏情况下所需的时间是指数级的,尽管这种情况极少见。此算法由 D. R. Smith 于 1984 年提出,他从理论上证明了有关平均时间的结论,这一结论与实际运行的几组随机产生的问题的结果一致。

图 10.6 给出了分支限界技术在串行处理机上的执行过程。此算法与一个子程序有关,这个子程序可以求出访问 N 个城市而且距离最短的一个排列。我们用记号 (1 2 3) 表示从城市 1 出发,经城市 2,城市 3,再回到城市 1,我们称此为一个回路,它的出发点也就

是它的最后终点。我们把像(1 2 3)(4 5 6 7)这样的一组回路称为城市的一个排列,每个城市在回路中仅出现一次。

一个排列并不一定是一次旅行。例如上面的排列中,从城市 1 出发,经城市 2,3 返回到城市 1,并没有访问其它几个城市。一次旅行应该将所有的城市都访问一次,显然,一次旅行是仅包含一个回路的排列,例如排列(1 2 3 4 5 6 7)是七个城市的一次旅行。

寻找最短距离排列的子程序就是在所有排列中找到城市与城市之间距离和为最短的排列。最小费用的排列给出了最短旅行的下界。旅行是一种特殊的排列,对于某个给定问题的最短旅行不可能比最短距离排列的距离更短了。

找最短旅行路径很困难,但找最短距离排列则相对容易些。第一次执行这个子程序只需 $O(N^3)$ 时间,随后的执行只是输入数据有一点差别,所以调用这个子程序只需增加 $O(N^2)$ 的工作即可得到结果。

图 10.6 显示了如何使用下界信息。图 10.6(a)给出了搜索树的一个结点,标着排列(1 2 3)(4 5 6 7),它的总距离为 151。通过运行最短距离排列算法得到这个总距离。为了避免只含一个城市的回路,设城市到它本身的距离为无穷大。由于(1 2 3)(4 5 6 7)不是一个旅行,所以最短旅行的距离一定大于或等于 151。最短旅行和这个排列的每个回路至少有一个分支不同。因此,不失一般性,不妨考察最短的回路,在这个例子中就是(1 2 3)回路。

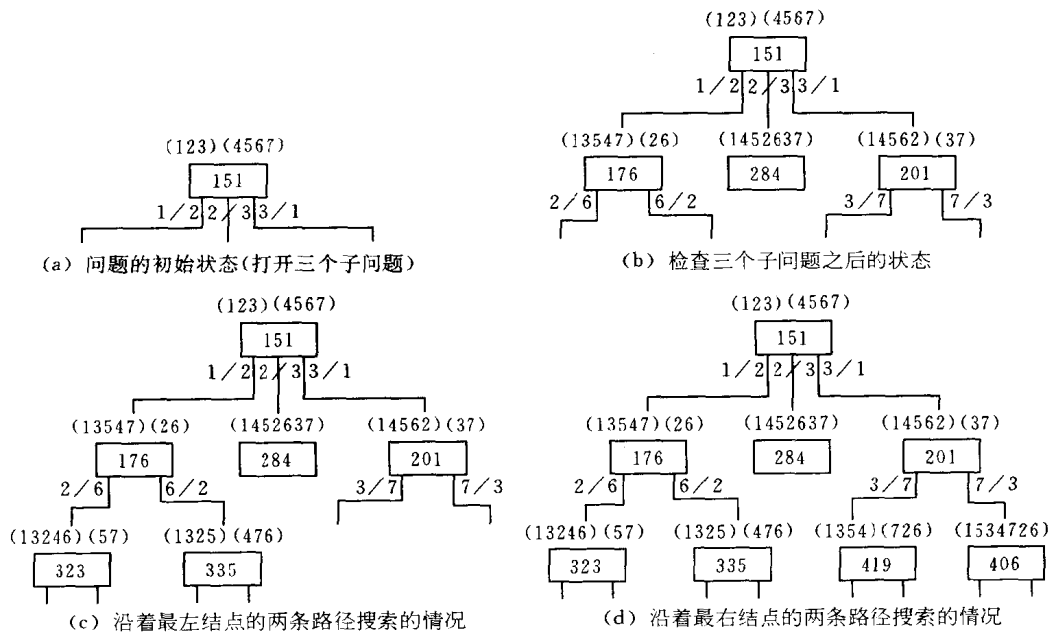


图 10.6 分支限界法解旅游商问题

最短旅行至少有一条路不同于上述回路,也就是说,最短旅行可能不从城市 1 到城市 2,或不从城市 2 到城市 3,或不从城市 3 到城市 1,这三种可能在图 10.6(a)中用三条离开初始结点的弧表示。

因为这三条路至少有一条路不会在最短旅行中,所以可以引出三个子问题来研究,我

们要分别测试它们是否在最短旅行中。图 10.6(b)给出了经过这一步后的结果。

图 10.6(b)的最左边的结点是将城市 1 到城市 2 的距离设为无穷大的情况。在这一条件下,调用最短距离排列的子程序,它返回的最短距离排列为(1 3 5 4 7)(26),总距离为 176。这个排列中没有从城市 1 直接到城市 2 的路,因为此路已被定义为无穷大了。当城市 2 到城市 3 的路被定义为无穷大时,得到的最短距离排列正好是一个旅行路径,距离等于 284。虽然已找到了一个旅行,但不能确定它就是最短旅行。我们还需要证明此路径是最短或找出另一条距离更短的路径来。

当城市 3 与城市 1 之间的距离被定义为无穷大时,最短距离的排列为(1 4 5 6 2)(3 7),总距离为 201。现在可以肯定不会有比 176 更短距离的旅行了。但如果将左右两个子问题继续进行搜索,或许能找到距离比 284 更短的旅行路径。

现在考虑最左边的结点,我们可以将排列的最小回路分开,将从城市 2 到城市 6 的路径或从城市 6 到城市 2 的路径设为无穷大,这是两条不同的路径。

同样,最右边的结点可将城市 3 和城市 7 的回路分开。这样,有四条路径需要继续搜索。图 10.6(c)和(d)给出了沿着这四条路进行搜索的情况。沿着最左结点搜索可得到两个新的排列,距离分别为 323 和 335。最左边的排列是(1 3 2 4 6)(5 7),它是不含城市 1 到城市 2 直接通路和城市 2 到城市 6 直接通路的最短距离的排列。

图 10.6(c)和(d)给出了沿着这四条路进行搜索的情况。沿着最左结点搜索可得到两个新的排列,距离分别为 323 和 335。最左边的排列是(1 3 2 4 6)(5 7),它是不含城市 1 到城市 2 直接通路和城市 2 到城市 6 直接通路的最短距离的排列。

此时最右结点成为最小界结点。沿着这一分支进行搜索,可得到两个新的排列,它们距离分别为 419 和 406。后一个排列正好是一个旅行路径。至此,没有一个排列或路径的距离比 10.6(b)中 284 更短。因此,(1 4 5 2 6 3 7)是最短旅行路径,此问题得解。

虽然上例并不一定是典型的实际问题,但仍可以看到分支限界算法的作用。扩展一个结点所花的时间为 $O(N^2)$,如果要检查此结点的所有后裔后才能找到一个旅行路径,那么找到一个旅行路径要花费很长的时间。如果某个结点的界很高,这条搜索路径看来没有多大希望,就干脆放弃沿着此结点的搜索。

在图 10.6 中,7 个城市共有 $6! = 720$ 条不同的旅行路径,限界法可将 718 条路径免于考虑。当然我们不能保证此算法在通常情况下都如此有效。D. R. Smith 于 1984 年已经证明求最短距离排列的平均次数为 $O(N \lg N)$,而最短距离排列算法的平均时间为 $O(N^2)$,所以算法总的平均时间为 $O(N^3 \lg N)$ 。

因为 Smith 的结论是假设问题符合均匀分布,所以,对于那些分布不均匀的问题该结论可能不成立。下面我们假设某一类问题满足 Smith 的假设,现在来考虑如何利用并行性来获得高效率。

假设每个旅行路径都从城市 1 出发,那么图 10.6 的搜索树通常情况下应有 $(N-1)!$ 个叶结点,每个叶结点表示一种可能的旅行路径。因此,如果平均分支因子正比于 N 的话,那么搜索树的深度为 $O(N)$ 。如果平均分支因子不大于一个与 N 无关的常量的话,那么搜索树的深度为 $O(\lg N!) = O(N \lg N)$ 。如果深度为 $O(N)$,那么访问 $O(N \lg N)$ 个结点平均要测试 $O(\lg N)$ 条并行路径。这意味着我们可以用多处理机并行地测试 $O(\lg N)$ 条路径。假如用 $O(N)$ 台处理机去处理所有可能的子问题,获得的加速比只等于 $O(\lg N)$,这与 Karp-Miranker 问题的加速比相同。

对图 10.6(b)来说,可以用两台处理机同时处理最左边结点的两个子问题,也可以用两台处理机同时处理最右边结点的两个子问题。然而,如果最左边结点返回的旅行路径的距离小于 201(201 是最右结点的距离),那么最右边结点所做的一切计算都白费了。这与 Karp-Miranker 浪费情况类似。

如果搜索树的深度为 $O(N \lg N)$,那么对多条路径并行地搜索其结果未必良好。在这种情况下,若用串行方法搜索,平均只有一条路径处于活跃状态。如果并行地搜索多条路径,那么几乎可以肯定除了一条路径外,其它所有路径的计算都白费了。

所有处理机都用来求一个结点的最短距离的排列是充分利用了并行性。一个有效的途径是仅仅测试那些纯串行算法才测试的结点,这就可以保证不会出现测试无用结点的浪费。

在测试一个结点的过程中,为了尽快找到最短距离的排列,应尽可能使用所有可使用的处理机。使用的处理机数目随着系统结构的不同而不同,这主要与通信和变量的访问有关。

假设共享资源的冲突并没有使系统的性能下降太多的话,那么 Dijkstra 的最短路径算法在某些 N 台处理机的系统上运行可以获得 $O(N/\lg N)$ 的加速比。然而,加速比与系统结构有关。如果一个系统执行 Dijkstra 的最短路径算法可以产生 $O(N/\lg N)$ 或更高的加速比,还假设要解决的旅游商问题的统计分布满足 Smith 的假设,那么就可以快速、高效地解决旅游商问题。

在这个例子中,关键是沿着多条路径并行搜索并不是相互独立的,这可能出现浪费,而只沿着一条路径查找,则有利用并行性的机会。宁可选择一条有希望的路径并集中所有处理机进行查找,也不要将处理机分散在几条路径上。

10.4 串行算法到并行算法的转换

把多处理机系统付诸应用的一个主要障碍是需要专门为这种系统结构编写程序。在最坏的情况下,每个问题都要重新研究和重新设计与相应系统结构一致的算法。这对于需要几天甚至几个星期计算时间的问题是值得的,因为机时的大幅度减少补偿了人们在算法优化上所花的精力。但是,对于中等规模的问题来说,例如只需几十分钟计算时间的问题,将算法优化也许只会节省几分钟的计算时间,那就不值得了。因此,我们想用程序变换器,把串行程序自动地转换成能在多处理机上执行的并行程序。

把串行程序自动地转换成并行程序的一条途径是为标准的高级语言构造一个编译器,使得编译器的输出可以在多处理机上并行执行。有了这样一个编译器就能以最小的代价把现有的软件库转换成能在多处理机上执行的并行程序。但是,由于这些程序本来是串行地运行的,所以并行性不高,这样通过编译器转换为并行程序,其执行效率也可能比较低。

通过这种编译器所产生的并行程序代码一般没有通过人工转换得到的并行程序代码那样高效,能有人工转换的一半效率也就算不错了。即使效率只有十分之一,这种编译器仍不失为一种有用的暂时工具,它为并行系统生成程序提供了一条捷径。当然,这种低效

的转换最终必须经过手工或更好的编译程序重新转换,以产生满足实际要求的并行程序。

对于向量计算机来说,已有人在富士通向量处理机上和在 IBM 3900 向量处理机上做了大量的工作。他们设计的编译器所产生的程序代码几乎与最好的程序员编写的程序一样高效,而比经验不多的程序员编出的程序代码要高效得多。

为多处理机系统设计一个高质量的优化编译器是一项很困难的工作。多处理机的编译器工作比向量处理机的编译器工作落后,这是因为多处理机的程序转换问题非常复杂。向量处理机的编译器是要找出在许多处理机上同时进行同一操作的方法,而多处理机的编译器是要找出在多个处理机上在不可预计的时间内进行多种操作的方法。这两种类型的编译器的共同之处是它们都需要判明语句之间的相关性,以确定事件的调度顺序。

10.4.1 相关性分析

从串行程序中获得并行性的最有效的方法是把循环迭代在多台处理机上执行。在前面已介绍过这种技术,在程序 10.4 我们还引入了块大小(chunksize)的概念,说明一台处理机能够执行一组迭代而不只单个迭代。虽然还存在其它类型的并行性,而且它们可以被优化编译器所识别,但对典型的应用问题来说,并行地执行循环迭代是获得加速比的主要方法。下面重点讨论并行地执行循环迭代的方法。

一个优化的多处理机编译器有检查程序并行性的任务。实际上,编译器是检查出串行操作,那么剩下的所有操作都可以并行地执行了。为了把循环迭代转换为并行程序代码,编译器必须检查出相邻的迭代什么时候必须串行执行。作为相关分析的例子,考虑下面一个循环:

```
for i := 1 to N do
  A[i] := A[i-1]/B[i];
end do loop;
```

上述每一次迭代直接依赖于上一次迭代,因为上次迭代所写入的变量正好是此次迭代所读出的变量,即相邻的两次迭代出现了对同一个变量有先写后读的要求,我们把这种关系称为写/读相关。还有读/写相关和写/写相关的情况。读/写相关是相邻的两次迭代出现对同一个变量有先读后写的要求。写/写相关要求变量值最后由当前迭代写入而不是由上次迭代写入。

这个例中的相关性很容易由编译器检查出来,因为它仅局限于单个变量。有些例子的相关情况可能非常复杂,如下列语句所形成的迭代:

```
A[i] := A[C[i]];
```

在这种情况下,当 $C[i] < i$ 时,出现读/写相关;当 $C[i] > i$ 时,出现写/读相关。此外,如果 C 值在执行时才计算出,编译器就无法判断会出现哪一种相关,于是也就不能优化代码了。因此,只有当迭代中所有下标表达式以及循环变量都已知时,编译器才能检查出相邻迭代的相关性。若下标变量取决于程序执行情况,编译器就被迫假定存在相关性,否则,优化过程就会产生不能正确运行的程序。

检查相关性的一般过程是把一个循环迭代中需要读与需要写的变量名分别列表。若

一个变量名在两个表中同时出现,那么就可能存在读/写或写/读相关。所有写的变量都有可能存在写/写相关。编译器将进一步检查各种情况以确定是否确实存在相关。

当一个变量被两个不同的循环迭代写入时才会出现写/写相关。通常,这种情况是在循环中有两个不同的语句,如:

```
A[i] := B[i]/10;
A[i-1] := C[i] + B[i];
```

两个语句在同一迭代中出现,很明显,前一次迭代用 $i-1$ 作为下标值写 $A[i-1]$ 和 $A[i-2]$,导致对变量 $A[i-1]$ 的写/写相关。注意,此时我们假定下标在每次迭代中以步长 1 增值。若循环下标以步长 2 增值,就不存在因对 A 写入两个相继的值而引起的相关了。读/写和写/读相关与写/写相关一样也易于查出。

10.4.2 开发迭代的并行性

这一小节将说明如何利用相关性信息来指导串行程序到多处理机程序的转换。这里只给出几种技术,但用途很广,使一般程序能获得很高的加速比。还有许多有价值的技术,特别是各类具体程序的设计技术,没在这里讨论。

为了充分利用多处理机,需将不相关的迭代分解开。若不相关的迭代有足够高的 R/C 比值,这种分解能大大提高速度。有时又要将几个迭代合成一个较大的任务块,以通过提高 R/C 比值来提高有效性,尽管这样做会减低并行性,但最终还是改善了系统的性能。理想的情况是将相关迭代合成一个大任务,使整个程序是一个不相关的大任务的集合。

程序 10.10 是实现这种想法的例子。这一段程序在传统的串行计算机的程序中很容易找到。我们假定此程序不用 `do seq` 语句也不用 `do par` 语句。

程序 10.10 计算矩阵每一行的和

```
for i := 1 to N do
begin
  A[i,0] := 0.0;
  for j := 1 to N do
    A[i,0] := A[i,0] + A[i,j];
  end i Loop;
```

程序 10.10 中矩阵 A 是一个 $N \times N$ 矩阵,下标值由 1 至 N 。第 i 行的和存入 $A[i,0]$ 中。

迭代 (i,j) 和迭代 $(i,j+1)$ 之间存在写/读相关性,这是因为在这两次迭代中 $A[i,0]$ 又被读又被写。这要求相邻两次迭代必须串行地执行。

高级的编译器能够检测出下标 i 没有相关性,这样 i,j 循环可以互换,如程序 10.11 所示。为了保证第 0 列初始化,程序中有一个专门循环来置初值。从迭代 $A[i,j]$ 到迭代 $A[i+1,j]$ 不存在相关性,所以内循环的 N 个迭代可以并行地执行。对某个 i 值来说,它的 N 个迭代必须串行地执行,我们可以把这 N 个迭代合成一个任务。这样,整个程序分成 N 个大任务,它们之间是不相关的,可以并行地执行。

程序 10.11 计算矩阵每一行的和

```

for i := 1 to N do
  A[i,0] := 0.0;
  for j := 1 to N do
    begin
      for i := 1 to N do
        A[i,0] := A[i,0] + A[i,j];
      end j loop;
    end
  end

```

把内循环组合成几个中等大小的任务以获得更高的并行性也是可能的。例如，把内循环分成 K 个任务，每一任务计算 N/K 行元素的和，这 K 个任务可以并行地执行。对某个 i 值来说，变量 $A[i,0]$ 被 K 个任务共享， K 个任务最后都要把部分和存入 $A[i,0]$ 中。由于存在读/写冲突，所以把部分和加起来的操作必须串行地执行。

更巧妙的编译器能够查出每一行求和能以任何顺序做，这样将 K 个任务严格的串行执行变为并行执行。每个任务计算出一个部分和，在块的结束处把部分和加到共享变量上去。最后的加法可通过加锁/开锁或比较交换指令或取与加指令或其它类似的手段实现。 K 的选择应既有高并行性又有高的 R/C 比值。

上例要说明的关键思想是要观察算法所呈现的相关性。假若不存在相关性，执行的顺序可以随意改变。例中的下标的顺序改变在算法中是常见的。通过改变顺序，变换后的程序含 N 个并行任务，若采用分块技术，则并行任务为 KN 个，而不是 N^2 个串行迭代。变换后的程序不但并行性更高，而且它的 R/C 比值调整到使同步所需的开销为最低。

第二个例子是我们熟悉的关于求解泊松问题的内部循环。在程序 10.1 中，要修改的项依赖于它东、南、西、北的邻居。无论如何选择迭代，按行或按列，升序或降序，都会有读/写与写/读的冲突。因此，对这个程序来说，交换迭代的顺序没有什么好处了。

然而，仍有并行性可以开发。如果把矩阵的元素放在棋盘上，程序 10.1 中的迭代可以如下实现，即用四邻的红块的平均值来修改一个黑块，同样，用四邻的黑块的平均值来修改一个红块。因为没有红块直接依赖于其它红块，也没有一个黑块直接依赖于其它黑块，所以红块和黑块构成了两个不相关的变量集合。

因此，我们可以建立两个任务，一个任务是根据黑块修改红块，另一个任务是根据红块修改黑块。这两个任务根据下标进行分块，可分为更小的任务，块的大小选择既要有一定的并行性又要使 R/C 比值不能太小。程序 10.1 的迭代可这样实现：先修改黑块的值，然后修改红块的值，在每种颜色的修改结尾处要求有阻塞同步。每次修改在所有可用的处理机上实现。

采用红黑方块的迭代和程序 10.1 的迭代不完全一致。程序 10.1 当每个点被修改时，它的四邻中的两个已被修改过了。例如，第 i 行和第 $i-1$ 行已经是新的数据，但第 $i+1$ 行还没有被修改过。这样，每个点的西、北邻是新值，而东、南邻是旧值。这种迭代称为高斯-赛德尔迭代。

另一种方法是把整个矩阵中要修改的数值先全部计算出来，然后再修改。这种迭代称为雅可比迭代。雅可比迭代是用所有邻居的旧值。红黑块方法是用所有邻居的新值。在

一般情况下,这三种方法以不同速度收敛于同一个解。红黑块方法的收敛速度最快,因为它用新的数据迭代。雅可比迭代的收敛速度最慢,因为它用旧的数据迭代。高斯-塞德尔迭代的收敛速度介于两者之间。

一般来说,程序 10.1 所示的这种迭代计算也许收敛、也许发散、也许既不收敛也不发散而呈振荡状态。如果某个问题一定能够收敛,那么迭代时用的新数据越多,则收敛速度越快。因此,把程序 10.1 转换成在多处理机上用红黑块方法并行执行的程序将非常高效。

程序 10.1 采用红黑块方法在多处理机上实现非常理想。因为网格中一半的点可以并行计算,整个程序可分配到适当数目的处理机上执行,且块的大小可足够大,以使同步开销尽可能的小。多处理机系统处理网格中不规则边界和特殊区域要比向量计算机更容易更有效。

一个优化编译器能将一种迭代方法转换为另一种迭代方法的假设合理吗?如果优化编译器是一个通用的编译器,那么回答是否定的。因为有上百种转换,一个编译器无法实现这么多种转换。然而,如果一个编译器是专门用来编译一类特殊的应用问题的程序,例如偏微分方程,那么应该把实际程序中最有用的转换包括在编译器中。

10.5 同步并行算法和异步并行算法

多处理机的并行算法是可以同时运行和协同地解决给定问题的一组 K 个并发进程。如果 $K=1$,它被称为顺序算法。为了保证并行算法正确有效地工作以解决给定的问题,进程之间需要相互作用以进行同步和交换数据。因此在一个任务系统中,可能存在着进程与其它进程进行通信的一些点。这些点称为互作用点,它们把一个进程分成若干步。因此,在每步的结束处,该进程可以与其它某些进程进行通信,然后才开始下一步的计算。

由于进程间的相互作用,一些进程可能在某些时间被阻塞。必须让某些进程等待其它进程的并行算法叫做同步算法。由于进程的执行时间是变化的,它取决于输入数据和系统中断的情况,因此所有必须在给定点同步的进程要等待它们中最慢的进程。这种最坏情况下的计算速度是同步算法的一个基本弱点,它可能造成加速比和处理机利用率比期望值要差很多。

为了弥补同步并行算法的这个缺陷,提出了解决某些问题的异步并行算法。在异步算法中,进程一般不必互相等待,它们之间的通信是通过读取存放在共享存储器中动态地修改的全局变量来进行。然而,由于并发地访问存储器可能会产生冲突,这使得进程访问公共变量时会有小小的延迟。

构造并行算法的另一种途径是宏流水线。如果某个计算可以分成被称为步的若干部分,使得某一部分或某几部分的输出是另一部分的输入,那么这种计算可以使用宏流水线方法。图 10.7 是采用宏流水线的程序流程图。

10.5.1 同步并行算法

同步并行算法是由有下述性质的进程所组成的算法:即存在这样一个进程,它的某个进程步必须在程序的另一个进程已完成某个进程步以后才被激活。例如,采用最大程度分

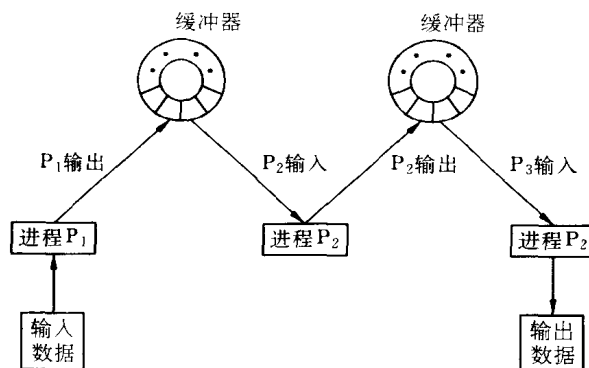


图 10.7 宏流水线中的程序流程

解的方法来计算矩阵 $Z = A \cdot B + (C + D) \cdot (I + G)$ 。通过建立三个进程 P_1 、 P_2 和 P_3 可以构成一个同步并行算法,如图 10.8 所示。

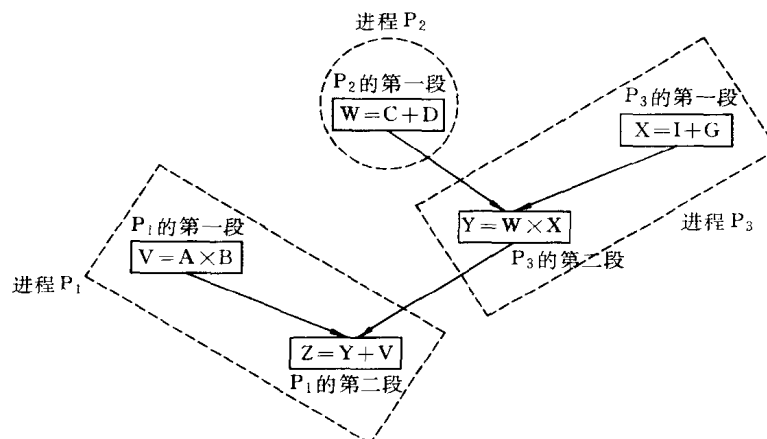


图 10.8 由进程步组成的同步算法例子

进程 P_1 、 P_2 和 P_3 分别由 2 个进程步、1 个进程步和 2 个进程步组成,如下所示。

程序 10.12 计算矩阵 $Z = A \cdot B + (C + D) \cdot (I + G)$

```

Var W,Y: shared real;
Var Sw,Sy: semaphore;
Sw=Cv=0;
Cobegin
  Process P1: begin
     $V \leftarrow A \times B$ ; (P1 的进程步 1)
    P(Sy);
     $Z \leftarrow V + Y$ ; (P1 的进程步 2)
  end
  Process P2: begin
     $W \leftarrow C + D$ ; (P2 的进程步 1)

```

```

        V(Sw);
    end
    Process P3: begin
        x ← I + G; (P3 的进程步 1)
        P(Sw);
        Y ← W + X; (P3 的进程步 2)
        V(Sy);
    end
Coend

```

程序 10.12 中 Cobegin-Coend 之间的进程可以并行地执行。P(s)和 V(s)分别表示对信号灯 S 的 P、V 操作。显然,进程 P₃ 的第二进程步在完成进程 P₂ 后才被激活。同样,除非进程 P₃ 的第二进程步已完成,否则进程 P₁ 的第二进程步不能开始执行。因此进程集 P₁、P₂ 和 P₃ 是同步并行算法。

由于进程步所花的时间是个随机变量,所以同步算法有个缺点,即某些进程在某个时间可能被阻塞,因此降低了算法的性能。我们考虑一个有 n 个进程的并行算法。假设该算法在一个有 n 台处理机的多处理机上运行,并且还假定算法的执行时间为 T_n 。在算法的执行期间,设 t_i 表示 i 个进程都在运行的总时间,即 $n-i$ 个进程被阻塞。因此, $T_n = \sum_{i=1}^n t_i$ 。假定该算法可以在单机系统上运行且运行时间为 $T_1 \leq \sum_{i=1}^n i \cdot t_i$ 。因此, n 台处理机的系统的加速比 S_n 的界为:

$$S_n = \frac{T_1}{T_n} \leq \frac{\sum_{i=1}^n i \cdot t_i}{\sum_{i=1}^n t_i} \quad (10.1)$$

显然 $S_n \leq n$ 。

现在考虑执行一组进程,其中需要同步 n 个相同的进程步,且第 i 进程步的执行时间是随机变量 t_i 。由于所有进程步是相同的,所以 t_i 服从平均值为 t 的相同分布的随机变量。 n 个进程步的同步意味着直到所有 n 个进程步都完成之后才能激活任一新的进程步。因此,任何经同步的进程步所花时间的期望值为随机变量 $T = \max_{1 \leq i \leq n} \{t_i\}$ 的平均值,而不是 t 。一般, T 大于 t 。比率 $T/t = \lambda_n$ 是同步 n 个相同进程步的惩罚因素。如果惩罚因素大,同步算法的性能就大大降低。加速比 S_n 和惩罚因素 λ_n 用来表示同步算法的性能。

同步迭代算法是同步并行算法的一个例子。实际上,大量问题可以采用迭代方法求解。例如,可用牛顿迭代法计算函数 f 的零点:

$$X_{i+1} = X_i - f'(X_i)^{-1} f(X_i) \quad (10.2)$$

其中 $f'(x)$ 是 $f(x)$ 的导数。用迭代方法解线性方程组,其解有以下形式:

$$X_{i+1} = AX_i + b \quad (10.3)$$

其中 X, b 是大小为 n 的向量, A 是 $n \times n$ 的矩阵。

一般可以用迭代函数来描述迭代算法,迭代函数为:

$$X_{i+1} = \varphi(X_i, X_{i-1}, \dots, X_{i-d+1}) \quad (10.4)$$

其中 $X_i \in R^n$

许多问题可以用方程(10.4)这种迭代函数来描述,如求解微分方程的松弛法、线性和非线性方程组的求解以及求函数的极值。多处理机分解迭代算法的一种方法是开发迭代函数内在的并行性。在 φ 的所有可能的分解中,人们总是试图得到若干个大小相同的并行任务或至少复杂性相同的并行任务,这样使得这些任务的执行时间互相独立并服从相同的分布。我们可以看到,在方程(10.2)的迭代过程中,求 x_i 的 $f(x)$ 和 $f'(x)$ 的值可以并行地进行。在方程(10.3)的矩阵迭代过程中,向量 X_{i+1} 的所有各个分量可以同时计算。在多处理机上实现方程(10.4)的另一策略是利用进程速度的波动。在这种情况下,使用多个进程并行地计算同一个函数,这样,首先得到计算结果的进程所用的时间比平均时间少。

10.5.2 异步并行算法

每个异步并行算法有一组所有进程都能访问的全程变量。当一个进程完成一个进程步时,它要读取某些全程变量。进程根据读到的变量值和上个进程步刚得到的结果,修改全程变量的子集,然后激活下一个进程步或结束进程本身。在许多情况下,为了保证逻辑上的正确性,把对全程变量的操作设计成对临界区的操作。

因此,异步算法的进程间通信是通过全程变量或共享数据进行的,且进程间没有像同步并行算法那种显性的依赖关系。异步并行算法的主要特点是任何时间进程都不必等待输入,而是根据全程变量当前包含的信息继续执行或者结束。然而,当许多算法都需要进入临界区时,进程可能会阻塞。

下面以与牛顿迭代求解函数 $f(x)$ 零点的算法相对应的异步迭代算法为例说明异步并行算法。为了方便,我们设三个全程变量 V_1, V_2 和 V_3 分别存放 $f(x), f'(x)$ 和 x 的当前值。例如,方程(10.2)的第 $(i+1)$ 次迭代之后, $f(x_{i-1}), f'(x_{i-1})$ 和 x_i 分别修改成 $f(x_i), f'(x_i)$ 和 x_{i+1} 。假设计算 $f'(x)$ 的值比计算 $f(x)$ 的值的代价要高,那么如下的由两个进程 P_1 和 P_2 组成的异步迭代算法是比较好的。进程 P_1 修改变量 V_1 和 V_3 ,进程 P_2 修改 V_2 。程序如下所示:

程序 10.13 求解函数 $f(x)$ 零点的异步并行算法

```
function f, f';
var V1, V2, V3: shared real;
Cobegin
  Process P1: begin
    while (termination criteria S not satisfied) do
      begin
        V1 ← f(V3); (P1 的第 1 步)
        V3 ← V3 - V2-1V1; (P1 的第 2 步)
      end
    end P1
  Process P2: begin
    while (termination criteria S not satisfied) do
      V2 ← f'(V3); (P2 的第 1 步)
    end P2
  Coend
```

从这个程序可以看到,一旦进程完成对某个全程变量的修改,它立即没有任何延迟地用有关变量的当前值接着进行下一次修改。假定迭代按进程 P_1 的第 2 步计算的顺序标号进行迭代,那么生成的迭代一般不满足方程(10.2)的递归关系。例如,如果变量的初值是 $v_1 = f(x_0)$, $v_2 = f'(x_0)$ 和 $v_3 = x_1$,那么每个进程每次迭代分步完成的顺序和时间可由图 10.9 所示的时间图来说明。时间图中分界线上的数字 i 表示该进程第 i 次迭代的开始点。那么对应这个图有:

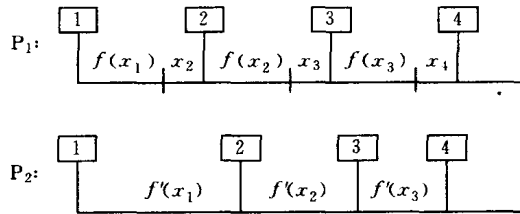


图 10.9 异步并行算法的时间图

$$x_2 = x_1 - f'(x_0)^{-1}f(x_1)$$

$$x_3 = x_2 - f'(x_1)^{-1}f(x_2)$$

$$x_4 = x_3 - f'(x_2)^{-1}f(x_3)$$

从上面给出的 P_1 和 P_2 的并行程序,我们可以得到进程运行一般遵循的递归关系如下:

$$X_{i+1} = X_i - f'(X_j)^{-1}f(x_i) \quad (10.5)$$

其中 $j \leq i$ 。因此,异步迭代算法产生的迭代不同于顺序算法或同步迭代算法产生的迭代。由于进程速度波动的原因,推导出关于序列 $\{x_i\}$ 特性的一般理论是很困难的。此外,由于异步迭代算法产生的迭代一般不满足方程(10.2)所示的确定的递归关系,所以要推导出关于收敛条件和收敛速度的一般理论是很困难的。

为一般的迭代进程(方程 10.4)设计一个异步迭代算法需要定义某个全程变量集合 $V[1], V[2], \dots, V[n]$ 的标识符,这样可以把每个迭代步看作是用 V_i 的老值计算它们的新值。这里一般要对 V_i 进行选择,使得修改每个 V_i 成为一次迭代的主要工作。例如,考虑方程(10.3)的矩阵迭代。在这种情况下,可选择 $V[i]$ 作为向量迭代中分量规模相等的段。选定 $V[i]$ 后,异步修改 $V[i]$ 的并发进程可以定义如下:假定向量 V_i 和 b 各有 n 个元素,把全程变量的集合 $\{V[1], V[2], \dots, V[n]\}$ 分割成 P 个子集,每个子集的大小是 $S = n/p$ (假定 n 可以被 P 整除)。第 K 个进程修改子集 $V[(K-1)S+1], \dots, V[KS]$, 其中 $V[j]$ 表示向量 X_i 的第 j 个分量的当前值。下面是一个包含识别一组全局变量的 P 进程算法,用来求解方程(10.3)表示的线性方程组。

程序 10.14 求解方程(10.3)表示的线性方程组算法

```

var V[1 : n]; shared real;
parfor k=1 until P do
  for i=(K-1)S+1 until KS do
    begin
      var acc; real;
      var A[(K-1)S+1 : KS, 1 : n]; real;
      var b[(K-1)S+1 : KS]; real;
      acc ← 0.0;
      for j=1 until n do
        acc ← acc + A[i, j] * V[j];
      V[i] ← acc + b[i];
    end
  end
end

```

上面的异步迭代算法要求迭代函数 φ 内部具有并行性。不利用迭代函数 φ 内在的并行性而是通过构造一个异步并行算法来加快迭代进程(方程 10.4)是可能的。这种算法称为简单异步迭代算法,它总是产生与顺序算法相同的迭代顺序。通常这种算法不是通过分担工作来达到加速,而是利用求值时间波动的特点达到加速。下面是一个简单异步迭代算法的例子,它由 K 个相同的进程 P_1, \dots, P_K 组成,每个进程用计算时迭代变量的最新值来计算迭代函数 φ 的值。在下面给出的并发程序中, i 和 X_i 是全程变量,而 j 是进程的局部变量。变量 i 的值是最近计算得到的迭代值的下标,因此,“if”语句是作为一个临界区来执行的。

程序 10.15 简单异步迭代算法例子

```
var s: semaphore; initial s=1;
var i, x[1:n]: shared: real;
par for i←1 until K do
begin
  while termination condition S not satisfied do
  begin
    j←i+1;
    X[j]← $\varphi(x[j-1], x[j-2], \dots, x[j-d])$ ;
    P(s);
    if i<j then i←j;
    V(s);
  end
end
```

简单异步迭代算法的主要优点是通用性好。这种算法并不仅仅局限于数值迭代进程。实际上它可以用来加快任何进程序列的执行。当进程分解有困难时,这些算法特别有吸引力。然而也存在一些缺点。首先在算法中需要用到临界区。第二,除非由于系统原因使计算时间的波动很大和随机变量 t 的变化系数(t 是一个进程计算一次迭代函数 φ 值所需要的时间)很大,否则算法的加速比将非常有限。

10.6 并行程序设计语言及其实现方法

10.6.1 并行程序设计语言的特点

计算机语言是计算机的灵魂,各种丰富多彩的应用问题都要用合适的计算机语言来描述。在传统的单处理器计算机世界中,我们熟悉的计算机高级语言有 C 语言、C++ 语言、Fortran 语言、Pascal 语言等,人们使用这些语言进行程序设计,开发计算机应用,使计算机在社会生活的方方面面发挥作用。

人们不断增长的需求和计算机技术的发展要求对计算机处理能力不断增高,单处理器计算机已经不能适应当前的这种形势,于是并行计算机系统,包括多处理机、机群系统等应用越来越广泛。

然而传统的计算机语言只具有顺序描述能力,即说明一个操作接着一个操作过程的能力,而不具备描述并行性的能力,也就是说明一个操作与另一个操作同时发生的能力,它已经不能满足计算机并行处理的要求,并行程序设计语言也就应运而生了。

因此,并行程序设计语言要具有描述程序并行性的能力。针对并行系统,一般而言,程序并行性分为控制并行性和数据并行性。控制并行性指两个不同操作可同时进行,数据并行性指对不同的数据同时执行同一操作。

10.6.2 并行程序设计模型

为了更好地说明并行程序设计语言,我们先说明一下程序设计模型。程序设计模型是一种程序抽象的集合,给程序员提供了一幅计算机硬件/软件系统透明的简图。在顺序程序设计中,常用的程序设计模型针对单处理器计算机,主要是结构化程序设计与面向对象程序设计。并行程序设计模型则是专门针对多处理机、向量机、大规模并行计算机以及机群系统而设计的。

并行程序的基本计算单位是进程,它与有关代码段执行的操作相对应。进程的粒度在不同的程序设计模型和应用程序中是不一样的。程序是进程的集合。并行程序设计的基本问题主要集中在相同或不同处理机并行进程的规范说明、创建、挂起、再生、迁移、终止、同步以及数据交换等方面。因此,并行程序设计具有多种模型,主要包括共享变量模型、消息传递模型、数据并行模型、面向对象模型等。

1. 共享变量模型

共享变量模型用限定作用范围和访问权限的办法,对进程寻址空间实行共享或限制。简单地说,就是利用共享变量来实现并行进程间的通信。为了保证能有序地进行 IPC,可利用互斥特性保证数据一致性与同步。共享变量模型与传统的顺序程序设计有许多类似之处。程序员只需关心划分程序中可并行进程,而无需关心进程数据交换问题。共享变量的数据一致性,临界区的保护性访问由编译器与并行系统来维护。适合采用该模型的并行系统一般是共享存储多处理机,如对称多处理机(SMP)。

2. 消息传递模型

消息传递模型是指程序中不同进程之间通过显式方法(如函数调用、运算符等)传递消息来相互通信,实现进程之间的数据交换、同步控制等。消息包括指令、数据、同步信号等,所以程序员不仅要关心划分程序中可并行成分,而且还需关心进程间数据交换。消息的发送接收处理将增加并行程序开发的复杂度。但是它适用于多种并行系统,如多处理机、可扩展机群系统等,而且具有灵活、高效的特点。

3. 数据并行模型

数据并行模型是指将数据分布于不同处理单元,这些处理单元对分布数据执行相同的操作。数据并行程序使用预先分布好的数据集。运算操作之间进行数据交换操作。数据并行性既可以在 SIMD 计算机上实现,也可以在 SPMD 多计算机上实现。这取决于并行成分的粒度大小以及所采用的操作方式。数据并行操作的同步是在编译时而不是在运行时完成的。

4. 面向对象模型

面向对象模型是近几年随着面向对象技术发展而提出的。它基于消息传递,但是并行处理单位是对象。在这种模型中,对象是动态建立和控制的。处理是通过对象间发送和接收消息来完成。

并行程序设计模型的选择一般与实际并行系统的特点关系密切。共享变量模型具有编程简单、易于控制的特点,但若在多处理机、机群系统上实现则会导致系统开销较大,因此共享变量模型常常用于共享存储多处理机,而不用于多处理机、机群系统。消息传递模型具有灵活高效的特点,因此一般用于多处理机、机群系统,充分发挥并行系统的性能。数据并行模型中的 SIMD 模型可用于向量机、多处理机等并行计算机,而 SPMD 模型则可用于多处理机、机群系统,但相对而言缺乏灵活性。面向对象模型具有简洁灵活的特点,适合多种平台,但系统开销较大。

并行程序设计语言就在这些并行程序设计模型的基础上,提出对并行性的描述方法,并行单元间协同的描述方法,以及该语言适用的并行计算环境。

10.6.3 并行程序设计语言涉及的关键技术

在当前的各种并行系统中,并行程序设计语言涉及的关键技术一般都比较相似。并行程序设计语言都要解决并行进程的描述与管理问题,进程间数据分布、传递的描述与管理问题,以及进程间同步协同的描述与管理问题。机群系统由于具有各个结点相对较为独立、结点之间通信开销较大等特点,比较适合于中、粗粒度任务并行处理,因此,面向机群系统的并行程序设计语言针对这些特点往往采用消息传递模型和 SPMD 模型。无论是消息传递模型还是 SPMD 模型,在机群系统上,由于每一个任务处理结点都是一个完整独立的计算机系统,所以上述问题的解决有它自己专门的特点。

1. 进程管理

进程管理完成进程的基本操作,主要包括进程的创建、进程的消亡以及进程的激活等。在机群系统中,并行进程通常是结点计算机操作系统的进程或线程。

(1) 进程创建

并行程序设计语言的进程创建有两种方式:显式(explicit)和隐式(implicit)。在显式方式中,语言提供显式的进程创建机制,这通常是一些预定义的进程创建函数。用这种方式很容易在已有的顺序语言上增加并行机制,而不必修改语言本身,如 PVM 系统、PRESTO 系统。在隐式方式中,并行程序设计语言采用关键字等方法描述进程,经过编译处理,由运行时系统自动创建并行进程。这种方式更加自然灵活,但语言实现复杂。大多数并行程序设计语言采用隐式进程创建,如 Thread C 等。

在机群系统中,进程创建一般调用结点计算机操作系统的系统调用创建进程,例如,UNIX 的系统调用 fork()。

(2) 进程消亡

进程可以显式或隐式消亡。在隐式消亡方式中,当并行任务处理完毕时自动退出。在显式消亡方式中,进程在处理完任务后等待进程结束指令,由程序员显式终止进程。显式方式比较适合消息驱动的程序设计。在机群系统上,绝大多数并行程序设计语言采用隐式消

亡。一些并行面向对象语言如 ABCL/1 采用显式消亡方式。

在机群系统中,一般采用系统调用结束进程,例如,UNIX 的 `exit()`。

(3) 进程激活

进程可以在创建时激活,也可以在收到消息后才激活。一般的并行程序设计语言都采用创建时激活。

在机群系统中,根据结点操作系统特点,一般都是进程创建后就激活。

(4) 进程粒度

机群系统比较适合中粗粒度的并行进程,因此并行程序设计语言一般都提供中粗粒度进程描述手段。

2. 进程通信与同步

在机群系统中,由于采用网络互连,因此消息传递与数据分布是并行程序设计语言中非常重要的一部分。通信与同步描述和管理能力的高低在一定程度上决定了一种并行程序设计语言的好坏。

在并行程序设计语言中,通信通过进程间的数据传递完成。它包括三种类型的通信方式:同步、异步和未来(future)。

同步通信方式类似于远地过程调用(RPC),最易实现。在同步通信方式中,数据发送方和接收方必须等待会合(rendezvous),发送方在收到接收方的数据接收确认后,才继续执行,因而系统效率不是很高。同步通信方式可靠,易于验证,在进程间协同管理时常常采用。

异步通信方式消除了等待会合,数据发送方不必等待接收方数据接收确认就继续执行,因而提高了系统运行效率。但异步通信方式可靠性比同步通信方式差,难于调试和验证。因此,常用于数据传递,而在程序正确性保证方面采用同步方式。

未来通信方式介于同步与异步通信方式之间。消息发送总是立即返回,不必等待,因而增加了并行性。但返回的不是消息结果,而是一个未来变量(future variable),用以保存消息返回值。当发送方需要消息返回结果时,则进入与接收方的同步点。系统自动检测未来变量中是否已存有消息返回结果。如果有,则继续执行;否则发送者阻塞,直至结果返回。未来通信方式比同步方式、异步方式增加了系统运行开销。

在机群系统中,由于系统通信开销较大,因此大数据量传送一般采用异步方式,而进程间协同采用同步方式。

并行程序设计语言通常提供数据发送与接收以及进程同步的函数或关键字。机群系统利用网络通信协议实现这些函数或关键字语义。

10.6.4 并行程序设计语言的实现途径

开发设计并行程序设计语言一般有三种方法:一,设计一种新语言。新语言有比较强的并行性描述能力,但是兼容性不好,不易推广,因此现在采用该方法的系统较少。二,对现有的顺序语言加以扩展,提供并行性描述机制,该方法具有兼容性较好,编程简便等特点,现在常常被采用。三,不改变现有顺序语言,而是用户提供函数库、类库或并行化编译系统等方法实现并行程序设计。该方法简单灵活,易于推广。

针对这三种不同方法设计的并行程序设计语言,一般相应地采用下述几种编译器实现方法来完成并行程序设计语言的编译处理:一,设计新语言的编译器;二,利用通用编译器,加入预编译;三,使用通用编译器,链接并行函数(类)库;四,针对传统顺序程序,设计并行化编译系统。

机群系统由于结点计算机一般都带有常用语言编译器,如 C 语言、Fortran 语言等,因此其并行程序设计语言的实现往往采用上述的后三种编译实现途径。

1. 新语言编译器

新的并行程序设计语言是针对并行程序设计特点开发的语言,比较著名的有 Occam 与 Ada 语言。它们都是提出一套崭新的语言文法,不仅包括并行任务、通信同步等的文法描述,还包括串行成分的文法描述,例如类型、过程、函数等。对于新设计的语言,人们要设计相应的新语言编译器。

由于新语言的编译器实现难度大,又常常针对一些专用系统,而人们常用的通用平台往往没有这些语言的编译器,这就导致采用这些语言开发的应用系统可移植性较差。另外,应用单位的新语言培训费用和过去开发的应用系统重开发费用都较高。所以,现在人们一般不设计和使用不具有通用性的新的语言和相应编译器。

特别是在机群系统上,因为每个计算机结点都是一个通用、完整的系统,所以开发新的语言和相应编译器未免对原有系统的利用不够充分,也不太经济。

2. 预编译处理

扩展传统语言实现并行程序设计是并行系统开发人员常用的方法。常常采用的编译实现方法是在传统语言的编译器基础上设计预编译器。通过预编译,将扩展语言的并行成分用传统语言加以实现,再通过传统语言编译器编译生成可执行代码。在机群系统上利用结点计算机原有的编译器,采用预编译技术实现并行程序设计语言,既提供较强的并行性描述能力,又比较实用,利于推广。作为扩展语言的基础语言常常是 Fortran, C, C++。例如:Concurrent C 语言与 Thread C 语言。

Concurrent C 是 AT&T 贝尔试验室的 Gehani 和 Roome 提出来的。他们在不改动原有 C 语言的基础上增加了对并行进程的说明,以及各个进程之间交互的说明。其中,并行进程是由关键字 process 来声明的,由 create 激活声明的进程,并且由关键字 trans 来声明交易(Transaction),由进程中 Accept 声明的服务代码处理交易请求。Concurrent C 的多个进程之间通过交易来相互通信。该交易是双向的,当一个交易建立后,申请服务的进程自动等待直至服务方完成请求的交易。这时交易的结果则传回给等待的客户。

Concurrent C 语言编译系统由预编译器 ccpp、标准 C 编译器、Concurrent C 运行时系统库等组成。预编译器 ccpp 将 Concurrent C 语言程序转换为 C 语言程序,其中一些关键字转换为 Concurrent C 运行时系统库函数,另一些转换为复杂的嵌入代码。然后通过标准 C 编译器将预编译器输出的 C 语言程序编译、链接成可执行代码。

Thread C 是清华大学在 C 语言基础上加以扩展、在机群系统上实现的并行程序设计语言。它提供了用于描述线程的代码、上下文的模板。此外,还提供用于线程间通信、线程迁移及线程管理的预定义的宏操作。在 Thread C 的编译系统中,预编译的工作是将用户的 Thread C 语言程序转换成高级语言程序(C 语言)。它由线程结构分析、前端编译、代码

生成三部分组成。在线程结构分析阶段,获得程序的结构信息,包括各线程声明、上下文结构等。前端编译则进行程序变换,将宏操作转换为内部操作。代码生成阶段生成一个线程管理框架。根据各线程的结构信息将转换后高级语言代码填入线程管理框架的相应位置。

3. 并行函数与类库

不改变传统语言,不改动编译器,而为程序员提供并行程序开发所需的函数库或类库,在编译生成可执行代码时,再将其链接进来。这是并行程序设计中新出现的动向,比较著名的有 PVM 系统。

PVM 为用户提供了在机群系统上开发并行程序的 C 语言和 Fortran 语言的并行函数库。用户直接用传统的 C 语言或 Fortran 语言设计并行程序,用户设计开发的每一个并行任务程序都是完整的 C 语言或 Fortran 语言程序。只是在需要描述并行特性的时候,加入并行函数。这样用户只要对过去用 C 语言或 Fortran 语言开发的系统稍加修改,嵌入并行函数,就能使之在机群系统上并行运行。

在 PVM 中,因为其编程模型是消息传递模型,所以 PVM 提供了许多并行任务管理、消息传递函数。在并行任务管理方面,有 `pvm_spawn` 函数创建、激活并行进程,有 `pvm_barrier` 函数进行多个进程同步。在消息传递方面,有任务间异步、同步消息传递,还有消息广播传递。其函数有 `pvm_send`, `pvm_recv`, `pvm_broadcast` 等。

由于机群系统的结点功能较强,直接利用已有编译器和系统环境开发的并行编程环境具有实现简便、使用方便、易于推广、生命力较强等特点。与 PVM 类似还有 MPI、Express 等。

清华大学在 PVM 的基础上实现了一个 C++ 语言的并行程序开发类库 `mpc++`, 它提供并行进程类、通信邮件类,以提高用户程序开发效率,减少错误。

4. 并行化编译系统

将传统顺序语言编写的应用程序自动并行化是并行程序设计领域的一个重要问题。它不开发新的程序设计语言,而是设计新的编译系统,使传统顺序语言编写的应用程序经过编译处理就能并行执行。这种方法使得用户能很方便地将过去的串行应用系统移植到并行系统上,培训费用和重开发费大大降低,因此具有良好的应用前景。由于科学计算领域中的应用系统一般都基于 Fortran 语言开发,所以并行化编译系统一般都针对 Fortran 语言。

目前,比较新的自动并行化编译系统有 UIUC 的 Polaris, Stanford 大学的 SUIF, 复旦大学的 AFT。

Polaris 系统采用过程间分析、符号数据相关性分析、数组私有化、归约识别、复杂形式的归纳标量识别等编译技术进行顺序程序并行化,另外对某些无法在编译时分析清楚的程序, Polaris 使用了运行时分析的方法对其进行并行化。

SUIF 是 Stanford 大学的编译系统平台。SUIF 也实现了过程间分析、符号相关性分析、数组私有化、归约识别等技术。此外, SUIF 还具有良好的开放性,研究人员可以很方便地在 SUIF 上试验新的编译技术,测试新技术与原有技术协同作用的效果。

复旦大学的 AFT 在采用的并行技术方面与上述两个系统相当,在稳定性方面还强于 Polaris,在功能上超越了传统的自动并行化系统。

另外,清华大学正在研究交互式并行化编译系统。由于全自动并行化编译系统在性能上仍有不尽人意之处,所以清华大学采用交互式并行化编译系统,允许用户在并行化过程中查看、询问、修改并行化结果,进一步优化并行代码,以期达到更好的并行效果。

10.7 小 结

到现在为止,我们仅呈现了多处理机系统结构现状的一小部分。但我们确信,所讨论的内容是多处理机系统结构中最重要的问题。开销和有效的并行性是两个最重要的问题。因此,实际的多处理机系统一般只有很少几台处理机。由 1000 台处理机组成的系统可望在几年内成为现实,但在这期间需要大量研究和解决有关效率的问题。多处理机系统的应用很大程度上依赖于下列几个问题的解决程度:

- (1) 消除 MYSPS 瓶颈;
- (2) 减少任务调度的开销;
- (3) 解决 Cache 存储器一致性问题或提供快速的局部存储器;
- (4) 串行程序到并行程序的转换;
- (5) 发掘有用的并行性和消除浪费的并行性。

随着这些课题研究的进展,多处理机系统会变得更具有吸引力,最终它将成为高性能系统结构的最佳选择。

前面的章节已经讨论了有关多处理机系统结构的主要技术问题。

1. 处理机带宽。由于系统中每台处理机都有为解决某个问题提供整个处理机带宽的可能,所以这一点是多处理机系统结构的一大优势。

2. 存储器带宽。存储器的可用带宽主要取决于存储器的多重访问机构。如果没有共享存储器,总带宽很高,因为它是单个存储器带宽的 N 倍。但有效的带宽较低,因为访问远程存储器时要通过一个或几个中间结点传递信息。如果有共享存储器,那么带宽和共享访问的实现方法有关。不同的实现方法,从共享总线到交叉开关,有不同的性能和开销。总线结构对于处理机较少的系统是最合适的。混洗交换网络或其它类似的多级互连网络对于较大的系统有吸引力,虽然开销增加了,但性能提高较大。对于处理机较少的系统来说,Cache 存储器是非常有用的。当处理机数量增至 8、16、32 或更大时,Cache 存储器的一致性问题在合理的开销范围内加以解决就困难了。因此,Cache 存储器一般只应用于局部变量、局部指令或其它不存在一致性问题的场合。访问不能快速存取的数据会占去共享存储器很大一部分带宽,同时它也是造成性能不能提高的原因。

带宽也受热点的限制。热点是指访问次数多于平均访问次数的那部分存储器区域。合并开关能把多个访问同一个变量的请求合并成一个请求,这样可以降低热点的“热度”。但合并开关处理热点问题是不是性能价格比最好的方法仍有争论。关于合并开关研究的成果对上百台处理机的系统的发展有重要的影响。

3. 输入输出带宽。多处理机系统能够提供与处理机数目成正比例的输入输出带宽。为了充分利用输入输出带宽,有必要采用与众不同的存储数据的方法。例如,一个文件分成多个段,这样多台处理机可以同时读写不同的段。假设每台处理机有独立的输入输出能

力,那么多处理机系统提供了很高的输入输出带宽。

4. 通信带宽。多处理机系统的通信带宽与采用什么样的互连结构有关。采用环结构或总线结构的系统,其通信带宽的成本比较低,只适合于 8~16 台处理机的系统。当处理机的数目多时,由于通信网络的竞争使系统的性能下降。为了支持上千台处理机,需要一种复杂得多的互连结构把处理机与存储器互连起来。

5. 同步。没有采用合并开关网络或类似的互连网络的多处理机系统的最大 MSYPS 值与处理机台数无关。因此,对中等或大型系统来说,最大 MSYPS 值是系统最严重的瓶颈。

采用合并开关技术可使系统的最大 MSYPS 值理论上与处理机台数成正比例关系。如果实际情况也是这样,那么合并开关将成为提高多处理机系统最大 MSYPS 值的主要机构。

6. 多用途。最通用的并行处理机是多处理机系统,因为每台处理机都能独立地操作。

上面讨论了多处理机系统的优点和缺点。优点是处理速度快,输入输出带宽高,灵活性大。缺点是同步、存储器带宽及通信带宽的问题。这三个问题向计算机系统结构设计者提出了挑战。在技术飞速发展的时代,新的方法几乎一夜之间就成为可能,而旧的方法有一夜之间就被废弃的可能。

习 题 十

10.1 解释下列术语

同步并行算法;异步并行算法;相关性分析;同步技术;并程序序设计模型;并行性粒度

10.2 某一指令序列的内部循环如下:

```
A[i] := B[i];
C[i] := A[i] + B[i-1];
D[i] := A[i+1];
```

① 指出此指令序列中哪些操作是相关的? 各个指令能否并行执行?

② 若第二条指令变为 $C[i] := A[i] + B[i]$,重复①。

③ 若第二条指令变为 $C[i] := A[i] + B[i+1]$,重复①。

10.3 一个程序的内部循环如下:

```
A[i,j] := A[i+1,j-1];
```

假设此条语句位于双重循环内,外部循环变量为 i ,内部的为 j 。设计一段循环控制程序,使之对变量 j 的操作集中(chunk together),而对 i 的操作分布到相互独立的处理器上,从而可以并行处理。

10.4 考虑做 n 次循环的 do par 语句:

① 一个带有局部与共享内存的多处理器系统,在 do par 语句被执行之前,所有的指令与数据均位于共享内存中。假设这些语句是相互独立的,无读/写、写/写、写/读冲突。设计一种能初始化这些循环的方案,使每个循环可独立地并发执

行;同时将此段程序的一个副本存于共享内存中以便各个循环访问。设循环变量为 i , 数组元素为 $A[i], B[i]$ 。为达到最佳性能,请确定在一个循环内,某一数据是应存于局部内存还是共享内存。

- ② 假设循环被串行地初始化的时间复杂度为 $O(n)$, 并行地初始化的时间复杂度为 $O(\log N)$ 。为多处理器的计算机写一程序,使之能初始化一个含有 128 个循环的 `do par` 循环,且时间复杂度为 $O(\log N)$ 。假设共享与局部内存的结构同①题,所有的处理器均可立刻初始化,且处理器的可用个数总是充分的。
- 10.5 题 10.4 中没有考虑处理器均处于忙状态时的等待任务队列问题。对一个规模为 $O(\log N)$ 的任务产生进程,在无空闲处理器的情况下,如何实现任务队列。
 - ① 设多处理器系统通过交叉开关访问共享内存,且等待任务排在单任务队列中。设并发任务数是可用处理器数的两倍,估计任务出入队列的开销。
 - ② ①中队列需要怎样的专用指令来实现。请描述这些指令的作用及操作数,并写出一个伪码段来说明这些指令。
 - ③ 考虑一个支持 Fetch-Add 的多处理机结构,重复①。
- 10.6 此题的目的是考虑 Barrier 操作的实现。若有一个多处理机,配有通过交叉开关访问的共享内存。
 - ① 写出实现 Barrier 操作的机器指令序列。设有 N 个处理机试图同时执行代码,估计一下你所设计代码的机器性能。说明你的代码在并发环境下正确执行的原因。
 - ② 在一个基于总线互连的多处理机上,重复①。
 - ③ 在一个支持 Fetch-Add 的多处理机上,重复①。
- 10.7 此题的目的是比较几种同步技术。设计一个长度为 N 的循环缓冲区(circular buffer),由子程序 put 与 get 控制缓冲区的输入和输出,实现中要避免死锁与活锁。
 - ① 使用 Test-and-Set 同步技术如何实现 put 与 get 操作,用高级语言和 Test-and-Set 来描述。
 - ② 使用 Increment 与 Decrement 代替 Test-and-Set,重复①。
 - ③ 使用 Compare-and-Swap 代替 Test-and-Set,重复①。
 - ④ 使用 Fetch-and-Add 代替 Test-and-Set,重复①。
- 10.8 此题的目的在于探讨 Compare-and-Swap 在链表队列中的使用。
 - ① 考虑本章中所描述的带头尾指针的任务队列,假设 DEQUEUE 操作不能与 ENQUEUE 同时进行,而 ENQUEUE 操作可同时进行,给出一个由 Compare-and-Swap 机制实现的 ENQUEUE 操作。
 - ② 给出一个由 Compare-and-Swap 机制实现的 DEQUEUE 操作的实现,你的实现是如何处理 DEQUEUE 后队列为空这种情况的? 另外你的实现能否使 DEQUEUE 与 ENQUEUE 同时执行?
 - ③ 利用 Double Compare-and-Swap 机制实现 ENQUEUE 与 DEQUEUE 操作,且使之能同时执行。

第十一章 计算机系统结构的新发展

11.1 数据流计算机

美国 MIT 实验室的 Jack Dennis 及其助手们于 1972 年首先提出了数据流(data flow)模型,并证明由此而设计的数据流计算机能够满足未来计算机理想结构的三个基本命题。这三个基本命题是:

- 第一,要获得很高的性能价格比;
- 第二,能跟上工艺技术进步的速度;
- 第三,在应用领域提供较好的可编程性。

数据流计算机的研究工作曾经被冷落过一段时间。近几年来,由于 VLSI 技术的迅速进步,为数据流计算机的发展提供了坚实的基础;因此,无论在硬件、软件,还是在理论研究方面,最近,数据流计算机都有了长足的进步,并且已经出现了一批能够实用的数据流计算机系统。

以下,首先介绍数据驱动(data driven)原理,着重分析数据流计算机与传统的冯·诺依曼控制驱动(control driven)计算机的不同;然后介绍数据流计算机的主要性能,重点分析数据驱动方式的优点和存在的主要问题,并提出今后设计数据流计算机所需要解决的一些基本问题,接着介绍数据流程图和数据程序设计语言,这是研究数据流计算机必须要了解的基础知识;最后介绍几台典型的数据流计算机,分析它们的系统结构和主要工作原理。

11.1.1 数据驱动原理

数据流计算机一般采用数据驱动方式工作,其工作原理与传统的冯·诺依曼计算机根本不同,它的指令不是在中央控制器的控制下顺序执行的,而是在数据的可用性控制下并行执行的。其基本原理可归纳为如下两点:

- (1) 当且仅当指令所需要的数据可用时,该指令即可执行。
- (2) 任何操作都是纯函数操作。

其中(1)说明指令的执行不受其它控制条件的约束,任何指令只要它所需要的操作数全部齐备且可用时,这些指令就可以同时执行;或者说,指令执行可以相互独立,操作结果可以不受指令执行顺序的影响,这就是数据流计算机所特有的指令操作的异步性和操作结果的确定性。

在数据流计算机中没有变量的概念,也不设置状态,在指令之间直接传送数据;因此,操作结果不产生副作用(side effect),不改变机器状态,从而具有纯函数的特点。所以,数

据流计算机通常与函数语言有着极其密切的内在联系。

由于上述数据驱动的基本原理带来了如下两个基本优点：

第一，使指令完全摆脱了外界强加给它的控制顺序，多条指令可以在数据可用性的驱动下同时并行执行。

第二，直接支持函数语言，不仅有利于开发程序中各级的并行性，而且有利于改善软件环境，缩短软件的研制时间。

在数据流计算机中可能有两种驱动方式，一种是数据驱动计算(data driven computation)，也就是上面介绍的数据驱动方式。只要某一个操作所要求的输入数据全部到齐而且可用时，该操作就可以立即执行，这是一种提前计算的策略。另一种是需求驱动计算(demand driven computation)，也称为需求驱动方式。只有当某一个函数需要某一个自变量时才驱动对该自变量的求值操作，这是一种滞后计算的策略。实际上，后者是按需求值，而前者是按数据可用性求值。

需求驱动方式与数据驱动方式相比可以减少许多不必要的操作，有助于提高处理机的工作效率。然而，需求驱动方式实现起来更为困难，因此，目前的大多数数据流计算机采用数据驱动方式。

11.1.1.1 串行控制流与并行控制流

在传统的冯·诺依曼计算机中采用串行控制流模型，而在并行多处理机系统中则采用并行控制流模型，两者的根本区别在于两个不同的程序表示方法中对计算顺序的控制方法不同。例如，要计算下面的这个算术表达式：

$$x = (a + b) \times (a - c)$$

图 11.1 表示用串行控制流计算函数 x 的过程。图中，用 $k, k+1, \dots$ 表示指令的地址，用大写英文字母 A、B、... 表示操作数和运算结果的地址，用小写字母 a, b, \dots 表示存放在相应地址内的数据。在指令执行过程中，有一条单一的控制流从一条指令传到下一条指令，指令所需要的操作数通过指令中给定的地址来访问。指令执行结果也通过地址存入一个共享的存储单元。

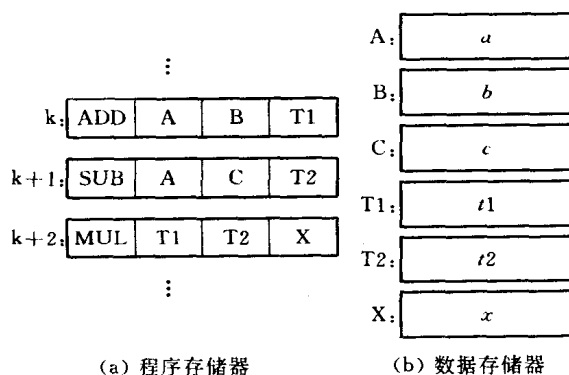


图 11.1 串行控制流模型

图 11.2 表示并行控制流模型,它采用操作 FORX 和 JOIN 来显式地表示并行性,并允许在同一时刻有几个控制流同时活动。并行控制流模型的一个关键技术是要有同步手段来处理数据之间的相关性,如图 11.2(a)中的 JOIN 操作。

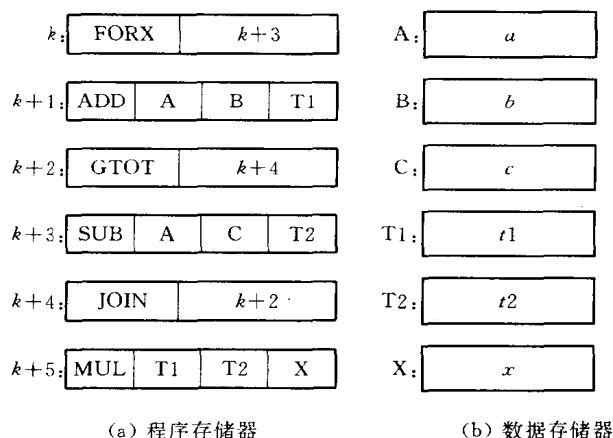


图 11.2 并行控制流模型

并行控制流计算机虽然摆脱了传统计算机单一控制流的束缚,允许有多个操作并行执行,但是,它仍然存在以下两个缺点:

- (1) 通常要用程序计数器 PC 来指明指令的执行过程。
- (2) 要通过访问同一个共享存储器才能实现在指令之间传送数据。

11.1.1.2 数据流计算机中指令的执行过程

在数据流计算机中,用数据令牌(data token)传送数据并激活指令,用一种有向图表示数据流程序。一条指令主要由一个操作符、一个或几个操作数及后继的指令地址组成;其中,后继的指令地址也可能有几个,它的作用是把本指令的执行结果送往需要这个数据的指令中。

图 11.3 表示函数 $x = (a+b) \times (a-c)$ 在数据流计算机中的计算过程。图中用符号“()”表示数据令牌所携带的操作数。

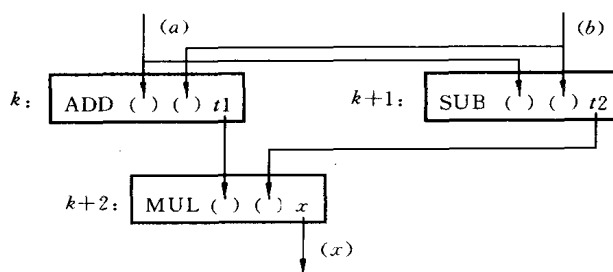


图 11.3 在数据流计算机中计算函数 $x = (a+b) \times (a-c)$ 时指令的执行过程

图 11.4 表示在指令执行过程中,数据令牌在指令之间传送的情况。在图中用“·”表示数据令牌。

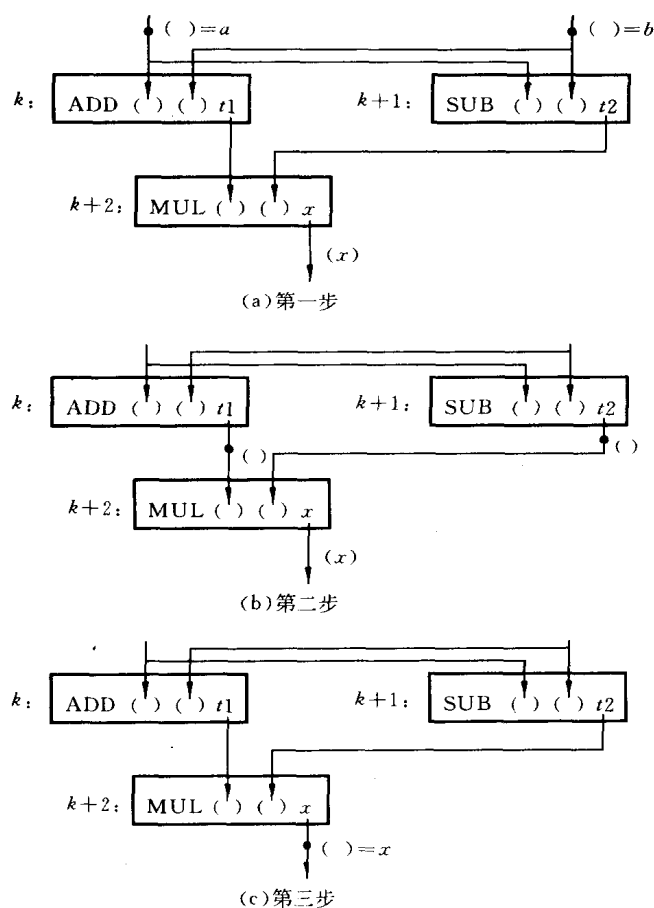


图 11.4 数据流计算机工作的瞬间状态图

数据流计算模型有以下两个特点:

(1) 数据(包括源操作数、中间结果和最终结果)作为数据令牌在指令之间直接传送。与控制流计算机中按照地址传送数据的概念截然不同。

(2) 只要执行指令所需要的源操作数的可用性都满足(即所有需要的数据令牌均到达)之后,指令即可以开始执行;因此,程序的执行过程只受到指令之间的数据相关性的限制。

根据以上两个特点,归纳数据驱动有如下四个性质:

(1) 异步性(asynchrony):只要执行本条指令所需要的数据令牌都已经达到,指令就可以独立地开始执行,而不必关心其它指令及数据的情况。

(2) 并行性(parallelism):可同时并行执行多条指令,而且,这种并行性通常是隐含的。

(3) 函数性(functionalism):由于计算过程中不使用共享的数据存储单元,因此数据

流程序不会产生诸如改变存储字这样的副作用(side effect)。也就是说,数据流运算是纯函数性的。

(4) 局部性(locality):在数据流运算过程中产生的操作数不是作为“地址”变量,而是作为数据令牌直接传送的。因此,数据流运算不会产生有长远影响的结果,它的运算结果具有局部性。

由于数据流运算具有异步性、并行性、函数性和局部性的特点,使得数据流计算机很适合采用分布方式实现。从这一点上看,也可以把数据流计算机看做是一种分布式的多处理机系统。

11.1.1.3 数据流计算机的指令结构

在数据流计算机中,一条指令主要由操作包(operation packet)和数据令牌(data token)两部分组成,如图 11.5(a)所示。

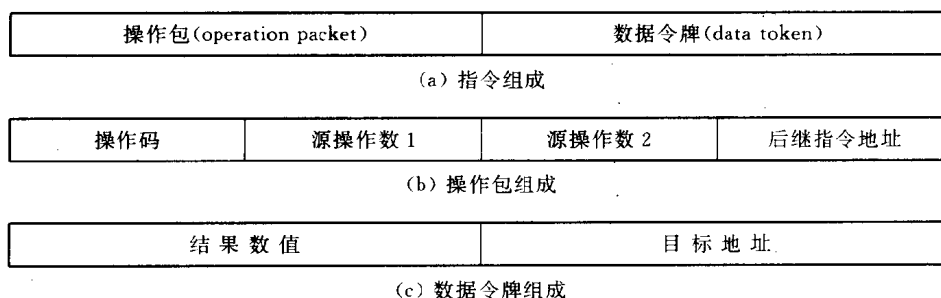


图 11.5 数据流计算机中指令的主要组成

图 11.5(a)中的操作包主要由操作码(operation code)、一个或几个源操作数(source data)及后继指令地址(next instruction address)等组成,如图 11.5(b)所示。这里的后继指令地址用来组成新的数据令牌,以便把本条指令的运算结果送往需要它的目标指令中。

数据令牌通常由结果数值和目标地址等组成。其中,结果数值就是上条指令的运算结果,而目标地址直接取自上条指令的后继指令地址,如图 11.5(c)所示。如果一条指令的执行结果要送往几个目标地,则要分别形成几个数据令牌。

在实际的数据流计算机中,指令的操作包和数据令牌中通常还包含有各种标志和特征信息等,这些将在最后一部分中结合典型数据流计算机进行介绍。

从图 11.5 所示的指令组成中可以看到,在数据流计算机中允许有多个操作包和多个数据令牌同时各个操作部件之间传送,允许有多条指令并行执行。因此,数据流计算机除了需要有一套能够并行执行多条指令的操作部件之外,还需要有一套高效的操作包和数据令牌传送机构。

11.1.2 数据流计算机模型

按照对数据令牌的不同处理方法,数据流计算机通常分为静态数据流计算机和动态数据流计算机两种。下面,介绍这两种数据流计算机模型,并且比较两种数据流计算机的特点,给出一种普遍化的数据流计算机结构。

11.1.2.1 静态数据流计算机模型

美国 MIT 计算机科学实验室的 Jack Dennis 和他的实验室研究人员首先提出了静态数据流计算机模型,这个模型如图 11.6 所示。

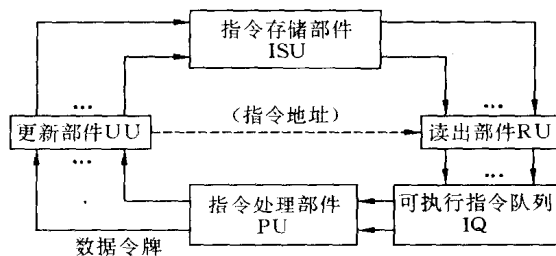


图 11.6 静态数据流计算机组成

在静态数据流计算机中,数据令牌沿数据流程图中的有向分支线移动到达操作符结点。当一个结点的所有输入分支线上的数据令牌都到达,且输出分支线上没有数据令牌时,就可以执行该结点的操作,或称为点火(firing)。另外还规定,在任何一个时钟节拍内在任何一条分支线上只允许传送一个数据令牌,这样做的好处是在数据令牌中可以不附加标号。数据令牌从一个结点传向另一个结点的时间顺序需要用另外一种令牌,即控制令牌来确定。

11.1.2.2 动态数据流计算机模型

动态数据流计算机的典型代表是 Watson 和 Gurd 等人在英国曼彻斯特大学研制的 Manchester 计算机及美国加州大学 Irvine 分校研制的 Arvinds 计算机。动态数据流计算机的模型如图 11.7 所示。

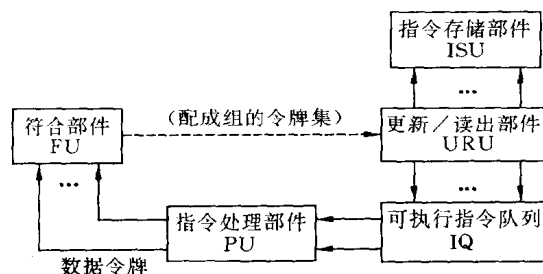


图 11.7 动态数据流计算机组成

在动态数据流计算机中,数据令牌可以带有标志,称为带标志的数据令牌(tagged data token)。因此,数据令牌在数据流程图中的有向分支线上流动时,同一条分支线上可以同时有几个数据令牌在移动。所谓标志,实际上就是在每一个数据令牌上附加一个标号,通过这个标号可以唯一确定令牌的状态。

动态数据流计算机可以最大限度地开发程序的并行性。例如,对于一循环程序,把数据令牌做上标志后可以同时展开几个相邻的不同次循环体并行进行计算,具体做法将在

下节中详细介绍。

11.1.2.3 静态与动态两种数据流计算机的比较

静态数据流计算机与动态数据流计算机的共同点是它们都有多个处理部件,可以独立、异步地执行多条指令,它们都依靠数据令牌来传送操作数和激活指令。两种数据流计算机的不同点是分别采用了两种不同的通信方式和两种不同的同步方式。

在图 11.6 所示的静态数据流计算机中,数据令牌开始存放在更新部件的输入缓冲器中,并通过该部件传送给指令存储器。在指令存储器中,按照数据令牌本身携带的目标地址部分的指示,把令牌中的操作数部分送到目标指令中。当一条指令所需要的数据令牌全部到达之后,这条指令就可以开始执行。指令本身首先由读出部件从指令存储器中读出,并送到可执行指令队列。一旦指令执行部件有空闲,这条指令就可以立即在指令执行部件中被执行。执行后指令把产生的结果与指令中给出的后继地址一起组成新的数据令牌,这个新产生的数据令牌又立即被送入更新部件的输入缓冲存储器中,由此完成一条指令的全部执行过程。

比较图 11.7 与图 11.6,不难发现动态数据流计算机与静态数据流计算机的主要不同是在动态数据流计算机中多了一个符合部件。在图 11.7 中,数据令牌首先被存放在符合部件的输入缓冲器中,这个输入缓冲器通常采用相联方式访问。符合部件把存放在缓冲器中的令牌配成对(或配成组,例如,执行一个二元运算,通常要两个数据令牌相符合),并且把配成对的数据令牌集送到更新/读出部件。更新/读出部件把送来的数据令牌对与使用这对数据令牌的指令相符合,并把执行指令所需要的操作数代入指令中,形成一条可执行的指令,送入可执行指令队列等待执行。只要某个处理部件有空闲,这条指令就可以立即执行。

由于动态数据流计算机的每个数据令牌上都带有特殊的标记,因此,可以通过符合部件把一个循环程序的不同次循环同时展开进行迭代,从而能够大幅度地提高程序的指令级并行度,缩短程序的执行时间。

从原理上分析,动态数据流计算机能够更加充分地开发程序中的并行成分,而且,由于中间运算结果直接代入下一条指令,而不像控制流计算机中那样要写入到共享存储器或通用寄存器中,从而减少了开销。但是,动态数据流计算机的复杂结构往往给设计者造成许多困难,例如,匹配部件通常成为影响并行度提高的一个瓶颈。另外,匹配部件中相联存储器的设计也是一个难点。

11.1.2.4 普遍化的数据流计算机结构

从图 11.6 和图 11.7 中可以看出,静态和动态两种数据流计算机都是一个环行流水线结构。如果把这两种数据流计算机抽象成为一个普遍化的结构,并加入输入和输出部分,成为如图 11.8 所示的一种普遍化的数据流计算机结构。

在一个环行流水线内有四个操作部件,指令存储器用来存放指令序列,处理部件专门并行地执行可执行指令,路径网络用来传送数据令牌,把令牌中所携带的操作数送入需要它的指令中,输入输出部分是数据流计算机与外界的接口。对于动态数据流计算机,数据

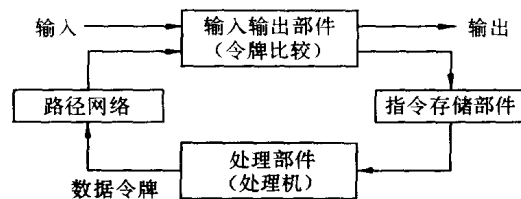


图 11.8 普遍化的数据流计算机结构

令牌的符合也在这部分内完成。

目前,大部分数据流计算机还仅作为主机的一个后端机来使用,而主机仍然使用传统的冯·诺依曼控制流计算机。通常,由主机完成程序的编辑、编译及各种输入输出等功能,而作为后端机的数据流计算机主要完成函数的计算工作。当然,对于数据流计算机的研究者们,最终目标是要设计出能够独立工作的数据流计算机。

实际上,一台能够实用的数据流计算机要由多个如图 11.8 所示的环形流水线组成。其中,路径网络划分为具有专门功能的令牌交换网络,指令存储部件也要分别由若干个存储模块组成。因此,实际的数据流计算机要由以上所介绍的数据流计算机模型扩充和改进而成,扩充和改进的方法多种多样,这样,就形成了多种不同类型的数据流计算机。后面将具体介绍各种数据流计算机的结构。

11.1.3 数据流计算机的性能分析

目前,数据流计算机及适用于数据流计算机的函数语言的研究工作还处在萌芽阶段。虽然数据驱动计算的思想早就有人提出,但是直到最近几年才研制出可供实用的数据流计算机模型机。现在就要对数据流计算机能够达到的性能做全面的评价还为时过早,然而,人们普遍认为,在这个领域应该继续研究和探讨。

11.1.3.1 数据流计算机的优点

数据流计算机在许多方面的性能优于传统的冯·诺依曼计算机。下面就这个领域里的研究人员们提出的优点进行归纳,其中的一部分优点已经得到模拟实验的验证。

(1) 高度并行运算

数据流方法本身就显示了操作的高度并行性,它不仅能够开发程序中有规则的并行性,还能够开发程序中任意的并行性。

在数据流方法中,由于没有指令执行顺序的限制,从理论上讲,只要硬件资源充分就能获得最大的并行性。已经通过程序验证,数据流计算机对许多问题的加速倍数随处理机数目的增加而线性的增长。

(2) 流水线异步操作

由于数据流计算机中的指令直接使用操作数本身,而不使用存放操作数的地址,因此,数据流计算机能够实现没有副作用的纯函数型程序设计方法,可以在程序的过程级和指令级充分开发程序中的异步并行性,可以把实际串行的问题采用简单的办法展开成并行问题来计算。例如,把一个循环程序的几个相邻循环体同时展开,把循环体内和循环体

间本来相关的操作数直接互相迭代,形成一条异步流水线,使一个循环程序内的不同次循环体能够并行执行。

目前,数据流方法已经广泛应用于流水线处理、数组处理和多重循环程序的处理中,取得了比较好的效果。

(3) 与 VLSI 技术相适应

从图 11.8 中可以看出,虽然数据流计算机的结构比较复杂,但是它的基本组成具有模块性和均匀性,其中的指令存储器、数据令牌缓冲器及可执行指令队列缓冲器等存储部件可以采用 VLSI 存储阵列均匀地构成。处理部件及信息包开关网络也可以分别用模块化的标准单元有规则地连接构成。因此,采用 VLSI 技术有希望研制出具有很高性能价格比的数据流计算机系统。

(4) 有利于提高软件生产能力

在传统语言如 Fortran、Pascal 等中,由于大量使用全局变量和同义名变量而产生副作用,给软件的生产和调试带来了很大困难。而在数据流计算机中,执行的是纯函数操作,使用函数程序设计语言来编程,从根本上取消了变量,取消了变量赋值机制。因此,数据流计算机彻底消除了巴科斯所说的冯·诺依曼赋值操作的瓶颈口。

用函数程序设计语言编写的程序符合程序设计方法学的要求,易读易懂。它良好的程序结构为程序的调试和验证提供了很好的基础。总之,数据流技术改善了软件的研制环境,可以提高软件的生产能力和软件的可靠性。

11.1.3.2 数据流计算机的缺点

数据流技术的反对者们指出数据流计算机在指令级并行性方面有许多潜在的问题。确切地说,上述这些数据流技术的优点实际上只有在理论化的数据流计算机模型中才具备。实际的数据流计算机为获得这些优点往往要付出巨大的代价,从而使得实际上的数据流计算机具有许多明显的不足之处。

(1) 操作开销过大

与传统的计算机相比,数据流计算机中各种操作的开销要大得多。为了提高程序并行度所付出的开销在相当广泛的实际应用领域里并没有得到应有的回报。

数据流计算机中的每条指令除了包含有操作码之外,还要提供两个操作数存储单元及存放多个后继指令地址的存储单元,所以,数据流计算机的每条指令都很长。很长的指令不仅要占用很多存储单元,而且使指令的存取变得复杂和费时。大致估计,数据流计算机的指令比传统计算机的指令所占用的存储容量和存取指令所用的时间要多 3~6 倍。有些动态数据流计算机,如 Manchester 数据流计算机,在指令中虽然不提供数据的存储单元,但又增加了许多标记,该计算机的指令长达 96 位,是巨型计算机 CRAY-1 的 5 倍。另外,数据流计算机的中间运算结果虽然不返回到共享存储器中,却又增加了访问相联存储器的匹配操作,这种操作与控制流计算机的寄存器—寄存器型指令相比,其操作的开销要大好几倍。

在数据流计算机中有大量中间结果形成的数据令牌在系统中流动,使信息的流动相当频繁,从而增加了冲突的可能性。为了减小冲突,要设置许多局部缓冲器,这样做不仅增

加了开销,而且延长了通信路径。虽然数据流计算机中各部件的异步操作能够形成一条宏流水线,使信息的通信障碍不会影响操作部件的利用率,但是过长的流水线需要大量的并行操作才能填满,这就使数据流计算机的应用非常局限。美国 MIT 实验室的 Arvind 等人对动态数据流计算机做了模拟,对于一台具有 64 个处理机的数据流计算机,为了填满流水线需要 640 个并行操作。

数据流计算机操作开销大的根本原因是把并行性完全放在了指令级上。在一台实际的计算机系统中,并行性可以分为很多级,例如任务级、作业级、进程级、过程级、函数级和指令级等。如果不适当地把高一级的并行性都依赖低一级的并行性来实现,往往要付出过高的代价。如已经相当成熟的数组运算就是一例。

操作开销大的另一个原因是完全采用异步操作,没有集中控制。在一台实际的数据流计算机中,其物理资源都是有限的,这就使得有许多可以同时执行的指令需要暂时存放在缓冲器中,并按照某种优先服务策略排队。另外,在整个系统中同时还传送和缓冲有大量的数据令牌、各种标志和应答信号等。所有这些都是异步的和随机的。为了解决这些异步操作和随机调度引起的混乱,需要花费大量的操作开销。

(2) 不能有效利用传统计算机的研究成果

数据流计算机完全放弃了传统计算机的结构,独树一帜。这样做一方面使它摆脱了传统计算机系统结构的束缚,具有活跃的生命力,另一方面却使它不能吸取传统计算机中已经证明行之有效的许多研究成果。例如,向量流水线已经被证明是一种性能价格比很高的技术,但在数据流计算机中由于采用异步操作和指令级操作并行,因而不能把这种技术吸收进来。

另外,数据流计算机企图摆脱困扰多处理机系统的许多问题,例如存储器按模块访问引起的冲突、复杂昂贵的互连网络、多进程之间的同步与通信等;但是在实际实现时往往又以新的形式回到这些老问题上。所以在数据流计算机的设计中认真吸取传统计算机的研究成果是非常有益的。

在软件方面,由于采用全新的函数语言,已经长期积累的大量软件成果不能继承。实际上,这一个基本问题如果不解决,数据流计算机将无法进入市场,无法与传统的计算机系统竞争。

(3) 数据流语言尚不完善

目前已经见到的数据流语言,如 VAL 及 ID 等都不完善,输入输出操作因为不属于函数运算,因此至今还没有引入到数据流语言中来。另外,数据流语言以隐含的方式描述并行性,而通过编译器来开发其中的并行成分,但是这种方式并不总是有效的。因此,数据流语言还需要进一步改进和完善。

在数据流程序中,由于引入了大量隐含的并行性,使得程序的调试工作变得非常困难。直到目前还没有一种好的解决办法。另外,对数据流计算机操作系统的研究还很少,在这一方面的研究还很不成熟。

11.1.3.3 数据流计算机设计中需要解决的几个问题

在分析了已经实现的几台数据流计算机模型之后,给出如下几个在数据流计算机设

计中需要解决的主要技术问题。

- (1) 研制易于使用、易于用硬件实现的高级数据流语言；
- (2) 研究程序如何分解、如何把程序模块分配给各个处理部件的算法；
- (3) 设计出性能价格比高的信息包交换网络,以实现资源冲突的仲裁和数据令牌的分配等大量通信工作；
- (4) 对于动态和静态数据流计算机,都要求研制智能化的数据驱动机构；
- (5) 研究如何在数据流环境中高效率地处理复杂的数据结构,如数组等；
- (6) 研制支持数据流运算的存储系统和存储分配方案；
- (7) 在广泛的应用领域里,对数据流计算机的硬件和软件作出性能评价,估计各种系统开销,包括开发、运行和应用开销；
- (8) 研究数据流计算机的操作系统；
- (9) 开发数据流语言的跟踪调试工具。

在以下的各个部分中将给出解决上述主要问题的途径,对其中的某些关键问题给出可供参考的解决办法。

11.1.4 数据流程图和数据流语言

数据流计算机的程序都是用数据流语言编写的,而最基本的数据流语言就是数据流计算机的机器语言,即数据流程图。本节分别介绍作为数据流计算机基础的数据流程图和数据流语言。

11.1.4.1 数据流程图

数据流程图是一种特殊的有向图(directed graph),它由有限个结点(node)集合以及把这些结点连接起来的单向分支线(unidirectional branch)组成。在数据流程图中,通过数据令牌沿有向分支线的传送过程来表示数据在数据流程图中的流动。当一个结点的所有输入分支线上都出现数据令牌,且输出分支线上没有数据令牌时,该结点的操作即可执行。

图 11.9 是计算函数 $x = (a+b) \times (a-c)$ 的数据流程图。图 11.9(a) 中的 a 、 b 、 c 三个圆点表示起始输入的三个数据令牌。图 11.9(b) 表示复制结点的操作执行之后的情况,它把数据令牌 a 复制为两个,分别送往加法操作结点和减法操作结点。图 11.9(c) 表示加

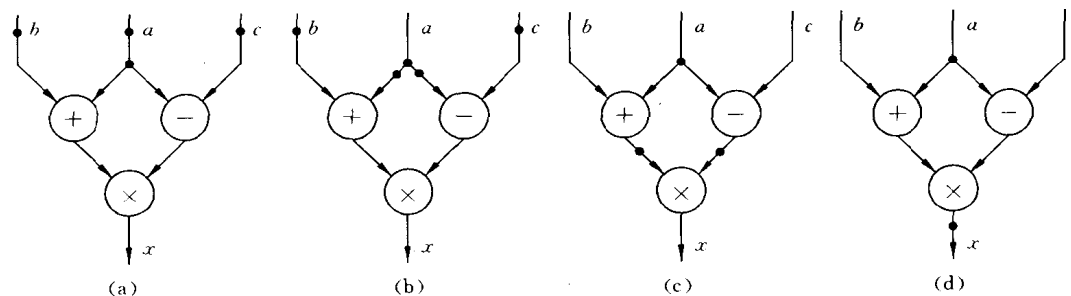


图 11.9 计算函数 $x = (a+b) \times (a-c)$ 的数据流程图

法和减法两个操作结点同时执行完成之后的情况。图 11.9(d)则表示乘法操作执行完成并输出最终结果 x 的情况。

J. B. Dennis 和 J. E. Rumbaugh 等人提出了一套用于数据流程图的各种操作符(即结点),并规定了相应的操作执行规则,现分别介绍如下:

(1) 复制结点

图 11.10 给出两种最常用的复制结点及其操作规则。其中,图 11.10(a)是数据复制结点,图中圆点和箭头都用实心的圆点和箭头表示。数据令牌 x 经过这个复制结点时执行复制操作,变成两个相同的数据令牌 x_1 和 x_2 。图 11.10(b)是布尔复制结点,复制结点中的圆点和图中的箭头都用空心的圆点和箭头表示。一个控制令牌 x 经过这个复制结点时执行复制操作,变成两个相同的控制令牌 x_1 和 x_2 。

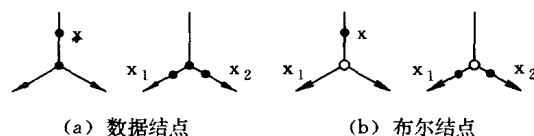


图 11.10 复制结点及其操作规则

(2) 运算结点

根据结点执行的操作功能可分为算术运算结点和布尔运算结点两种。常见的算术运算结点有“加”(+)、“减”(−)、“乘”(×)、“除”(÷)、“加 1”(+1)、“减 1”(−1)等,常见的布尔运算结点有“与”(∧)、“或”(∨)、“异或”(⊕)、“非”(N)等。

如果按照输入端的个数可以把结点分为单输入结点和多输入结点两种。如图 11.11 中的“+1”和“N”是单输入结点,而“+”和“∧”是多输入结点。

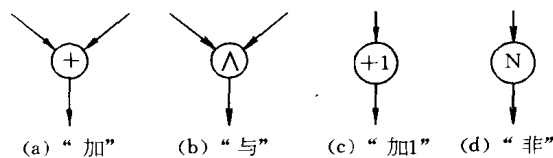


图 11.11 运算结点举例

图 11.11 仅仅是运算结点的几个例子,其中,算术运算结点中的箭头用实心的箭头表示,而布尔结点的输入和输出箭头都用空心的箭头表示。

(3) 常数发生器结点

常数发生器结点没有输入分支线,只有一条输出分支线。这种结点的功能是用来产生一个常数。图 11.12(a)是常数发生器结点的示意图。图 11.12(b)是常数“2”的结点及其执行操作之后产生数据令牌的情况,该数据令牌携带有常数“2”。

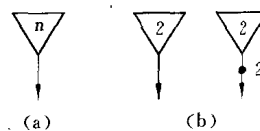


图 11.12 常数发生器及其操作规则

(4) 条件分支结点

在数据流程图中,条件分支结点主要用来控制数据令牌的传送时刻,它通过一条用

空心箭头表示的输入分支线来输入控制令牌。

常见的条件分支结点有四种,分别对应如下四种条件语句:

```

if true then  $x \rightarrow x_T$ 
if false then  $x \rightarrow x_F$ 
if true then  $x \rightarrow x_T$  else  $x \rightarrow x_F$ 
if false then  $x \rightarrow x_F$  else  $x \rightarrow x_T$ 

```

这四种条件分支结点及其操作执行规则分别如图 11.13 至图 11.16 所示。

图 11.13 是 T(true)门分支结点及其操作执行规则示意图。当输入分支线上的数据令牌 x 和携带有真值的控制令牌 T 都到达该结点,且输出分支线上没有数据令牌时,这个分支结点的操作将立即执行,执行的结果是把输入分支线上的数据令牌原样传送到输出分支线上。

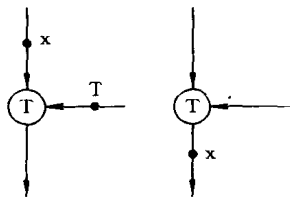


图 11.13 T 门分支结点及其操作执行规则

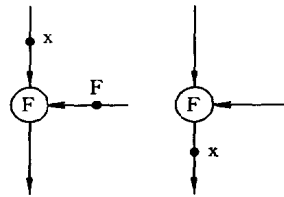


图 11.14 F 门分支结点及其操作执行规则

图 11.14 是 F(false)门分支结点及其操作执行规则示意图,其工作过程与 T 门分支结点类似,所不同的只是当输入分支线上的数据令牌 x 和携带有假值(false)的控制令牌 F 都到达该结点,且输出分支线上没有数据令牌时,这个分支结点的操作才执行。

开关分支结点用一个椭圆表示,这种结点有一条数据令牌输入分支线和两条数据令牌输出分支线,另外还有一条控制令牌输入分支线。图 11.15 是开关分支结点及其操作执行规则示意图。开关分支结点的操作执行规则是:当数据令牌输入分支线上出现数据令牌 x 时,如果控制令牌输入分支线上的令牌所携带的是真值(true),且 T 输出分支线上没有数据令牌时,则该开关结点执行的结果是把输入数据分支线上的数据令牌 x 传送到 T 输出数据分支线上,如图 11.15(a)所示;相反,如果控制令牌输入分支线上的令牌所携带的是假值(false),且 F 输出分支线上没有数据令牌时,则把数据输入分支线上的数据令牌 x 传送到 F 输出数据分支线上,如图 11.15(b)所示。

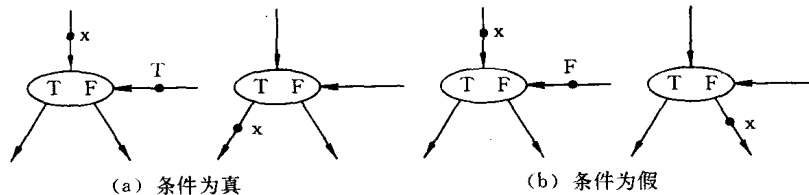


图 11.15 开关分支结点及其操作执行规则

图 11.16 是合并结点及其操作执行规则示意图。合并结点也用一个椭圆表示,但是它的作用与开关结点正好相反。合并结点有两条数据令牌输入分支线、一条数据令牌输出分

支线及一条控制令牌输入分支线,它根据到达的控制令牌携带的是真值还是假值决定把 T 输入分支线上的数据令牌还是把 F 输入分支线上的数据令牌传送到输出数据分支线上。

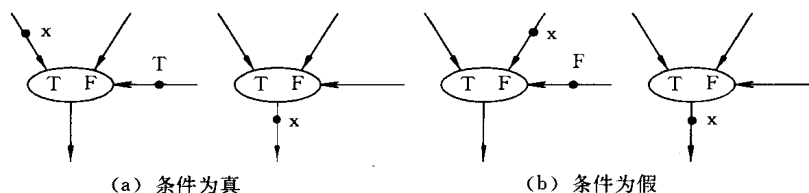


图 11.16 合并分支结点及其操作执行规则

(5) 条件操作结点

条件操作结点用一个菱形表示,它通常有一条或几条数据令牌输入分支线和一条控制令牌输入分支线。条件操作结点根据输入数据令牌所携带数值的某种特征(或几个输入数据令牌所携带数值的某种关系)作出判断,最终在输出分支线上产生一个控制令牌。这里所说的特征或关系通常包括: $x > 0$ 、 $x = 0$ 、 $x < 0$ 及 $x > y$ 、 $x = y$ 、 $x \leq y$ 等等,其中的 x 和 y 是指输入数据令牌所携带的数值。

图 11.17 是条件操作结点及其操作执行规则示意图。其中的图 11.17(a)是单输入分支线的条件操作结点,图中的 P 是判断真、假时使用的条件。图 11.17(b)是多输入分支线的条件操作结点。

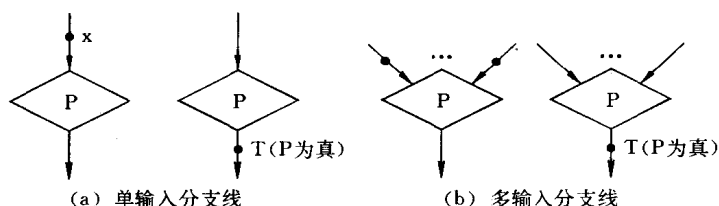


图 11.17 条件操作结点及其操作执行规则

图 11.18 是条件操作结点及其执行规则的举例。其中,图 11.18(a)是单输入分支条件操作结点的操作过程举例。在这个例子中,当输入分支线上出现数据令牌,而且输出分支线上没有数据令牌时,如果输入数据令牌所携带的数值 x 满足 $x > 0$ 的条件时,则在输出分支线上产生一个携带真值的控制令牌,否则在输出分支线上产生一个携带假值的控制令牌。图 11.18(b)是一个双输入分支条件操作结点的例子,其操作执行规则与单输入

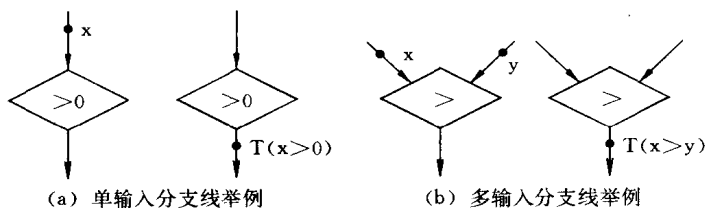


图 11.18 条件操作结点及其操作执行规则举例

分支条件操作结点类似,所不同的只是判断条件改为 $x > y$ 。

根据数据流程图的需要,可以利用上述这几种单功能的操作结点组合成功能复杂的多功能结点。下面举例说明这些单功能操作结点的使用方法。

例 11.1 利用上述单功能操作结点实现一般高级语言中的条件语句:

if true then G1 else G2

画出数据流程图,其中的 G1 和 G2 都是各自独立的数据流程图。

如图 11.19 所示,利用一个复制结点、一个 T 门分支结点和一个 F 门分支结点实现起始数据令牌的两路传送,它根据起始控制令牌所携带的是真值还是假值把起始数据令牌分别送往 G1 数据流程图或 G2 数据流程图。并且利用一个合并条件分支结点选择 G1 或 G2 数据流程图中的一个结果作为输出,选择的依据仍然是起始控制令牌携带的是真值还是假值。

例 11.2 利用上述单功能操作结点实现一般高级语言中的循环语句:

while P do G

或语句:

unit P do G

画出数据流程图,其中 P 是循环条件, G 是循环体。

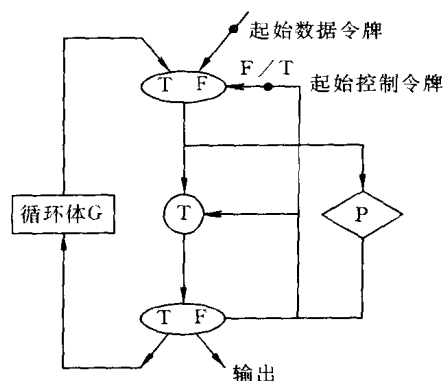


图 11.20 一种循环结构的数据流程图

根据循环结束条件 P 产生的控制令牌来控制循环的执行。最后用一个条件分支结点分配每次循环产生的结果数据令牌,如果循环还没有结束,则条件操作结点 P 输出为真值,通过条件分支结点把结果数据令牌分配给循环体 G,继续进行下一次循环;如果循环结束,则条件操作结点 P 输出为假值,通过条件分支结点把结果数据令牌输出到外部。

下面举一个实际的例子来说明如何用前面介绍的各种操作结点构成一个能够实用的数据流程图。

例 11.3 给定一个自然数 x , 求它的阶乘 $x!$ 。画出数据流程图。

如果用 C 语言来实现 $x!$, 可以描述如下:

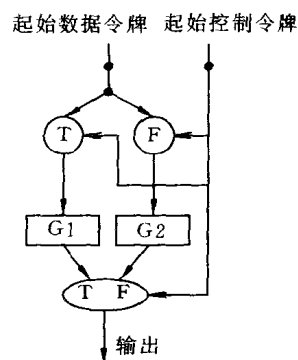


图 11.19 一种条件结构的数据流程图

如图 11.20 所示,为了使数据流程图中的循环体 G 能够开始执行,在一开始要输入一个起始数据令牌和一个起始控制令牌,并采用一个合并分支结点取得循环体 G 的输入数据令牌。在第一次循环开始时从外部输入数据令牌,而在以后的各次循环中都从循环体本身的输出端取得所需要的输入数据令牌。在第一次循环时,由一个 T 门分支结点控制起始数据令牌是否输入给循环体 G,这就是 while 和 unit 语句的区别,控制方法是起始控制令牌携带真值还是携带假值。另外,用一个单输入条件操作结点根据循环结束条件 P 产生的控制令牌来控制循环的执行。最后用一个条件分支结点分配每次循环产生的结果数据令牌,如果循环还没有结束,则条件操作结点 P 输出为真值,通过条件分支结点把结果数据令牌分配给循环体 G,继续进行下一次循环;如果循环结束,则条件操作结点 P 输出为假值,通过条件分支结点把结果数据令牌输出到外部。

```

main( )
{
    int x,i;
    scanf("x=%d",x);
    for(i=x;i>1;i--)
        x=x * i;
    printf("x! = %d\n",x);
}

```

图 11.21 是计算自然数 x 阶乘的数据流程图,从图中可以明显看出它有两个并行执行的迭代循环,一个是乘法操作循环,另一个是减“1”操作循环。为了使这个数据流程图能够开始执行,首先要从外部输入一个带有原始数值 x 的数据令牌和一个带有假值的起始控制令牌。通过复制结点把从外部输入的原始数据令牌复制成完全相同的两个数据令牌分别送往乘法操作循环和减“1”操作循环;然后在起始控制令牌的控制下,两个迭代循环分别同时开始执行。每执行一次循环,“ >1 ”条件操作结点就产生一个带有真值的控制令牌,在这个控制令牌的控制下并行执行一次乘法循环和一次减“1”循环。当“ -1 ”操作结点输出带有数值为“1”的数据令牌时,“ >1 ”条件操作结点就产生一个带有假值的控制令牌,在这个控制令牌的控制下停止乘法操作循环和减“1”操作循环,并把乘法操作结点产生的最后一个数据令牌作为最终运算结果输出到外部,在这个数据令牌中携带的数值就是最终运算结果 $x!$ 。

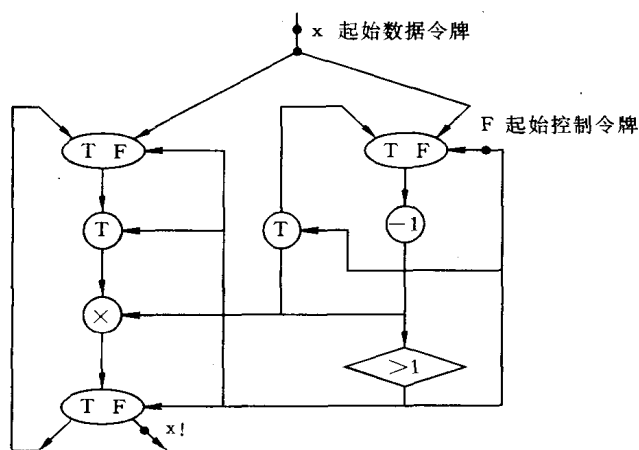


图 11.21 计算 x 阶乘的数据流程图

从图 11.21 所示的计算 x 阶乘的数据流程图图中可以很明显地看到,在数据流计算机中操作执行过程的异步性和充分的并行性。

表示数据流程图除了采用上面所介绍的结点分支线方法之外,还有另外一种表示方法,称为活动片表示法(activity templete)。在本书 11.1.1.2 节中计算函数

$$x = (a+b) \times (a-c)$$

的过程实际上就采用了活动片表示方法。图 11.3 和图 11.4 就是计算上面这个算术表达

式的活动片表示方法数据流程图。

在采用活动片表示法的数据流程图图中,组成数据流程图的基本单位是活动片。每个活动片相当于结点分支线表示法中的一个或几个操作结点。一个活动片通常由一个操作码域、一个或几个操作数域、一个或几个后继指令地址域及有关标志等组成。

比较表示数据流程图的两种方法,活动片表示法更接近于常规计算机中的指令系统,比较容易理解。实际上,在上一节中看到的数据流指令就是用活动片表示法表示的。在本节中介绍的结点分支线表示法更接近于一般的高级语言,具有可读性好、直观等优点,在程序设计中被广泛地应用。

数据流程图相当于数据流计算机中的机器语言。对于一般用户,如果直接用数据流程图编写程序是很不方便的。数据流程图与传统计算机中高级语言和超高级语言相比,不易被用户接受。因此,必须研制适合于数据流计算机的高级语言,这种高级语言应该能够用近似于自然语言的方式最大限度地描述隐含的并行性,并具有易读、易于理解和调试、维护方便等优点。

11.1.4.2 数据流语言

目前,数据流语言的研究还很不成熟,还没有形成像传统高级语言那样的规范版本。就已经出现的数据流语言而言,大致可以分为如下三类:

(1) 单赋值语言(single assignment language):包括美国加州大学 Irvine 分校研制的 ID 语言(Irvine data flow language)和美国 MIT 科学实验室的 Dennis 教授于 1979 年提出的 VAL 语言(value algorithmic language)。

(2) 函数类语言(functional language):比较著名的有美国犹他大学研制的 FP 语言(functional programming language)。

(3) 命令类语言(command language):目前,美国依阿华州立大学正在研制此类语言,并研制了把命令类语言转换成数据流语言的编译器。

下面仅就 VAL 语言的有关特点作简单介绍。

(1) VAL 语言易于开发程序中隐含的和显式的并行性,它提供了相应的语句结构来表达算法中的并行成分,从而能够高效地编写数据流程序。

(2) VAL 语言没有变量的概念,仅有数值的名称,它遵循单赋值规则,运算不产生副作用。因为并行度的检测无须经过全局分析,因此,它比命令类语言更易于开发程序中的各种并行性。

(3) 在 VAL 语言中提供了丰富的数据类型,它提供的基本数据类型有:整型、实型、布尔型和字符型等,结构类型有数组和记录等,而且允许数组和记录这两种不同的结构类型之间互相嵌套调用,嵌套的深度不受限制。

(4) VAL 语言是一种强类型语言,任何函数的自变量和计算结果的数据类型都要在函数的首部加以定义。编译程序在编译过程中能够很方便地检测出函数和表达式中数据类型发生的错误。

(5) 在源程序中不规定语句的执行顺序,没有 GOTO 之类的程序控制语句。语句的执行顺序不影响最终运算结果。

(6) 用 VAL 语言编制的程序具有模块化结构。实际上,用 VAL 语言编写的程序是一组模块的集合。每一个模块包含一个外部函数,这个外部函数又可以被其它模块调用。在一个模块内部往往包含有许多内部函数,这些内部函数仅仅供本模块内部调用。

VAL 语言还存在下述缺点:

- (1) 没有输入输出手段,特别是交互式输入输出手段。
- (2) 程序的表达式还不够自然和方便。
- (3) 实现的效率还很低。

11.1.4.3 数据流语言的性质

传统的程序设计语言是建立在冯·诺依曼系统结构基础上的,它以共享的存储单元为基础,以顺序控制方式执行语句。因此,控制流语言不能充分表达程序中的并行性,而且容易产生副作用。为了能够表达程序中的并行性,虽然人们开发了并发 Pascal、并发 Prolog、向量 Fortran 等语言,但是,由于这些语言是以顺序执行和多赋值为基础的,因此不能用到数据流计算机中来。

数据流语言与传统的程序设计语言相比具有如下特点:

(1) 并行性好

指令的执行顺序仅受程序中数据相关性的约束,而与指令在指令存储器中存放位置无关。因此,它能够以很自然的方式最大限度地表达程序中的并行性。

(2) 单赋值规则

根据单赋值规则,在所有语句的左边,同一个变量名只能出现一次。换句话说,要为任何一个重新赋值的变量选择另一个从前未曾使用过的新名字,而且在以后的所有引用中都使用这个新名字。请看下面两小段程序:

程序(1): $x = a - b$

$x = x + y$

$z = x - y$

程序(2): $x = a - b$

$x_1 = x + y$

$z = x_1 - y$

上述程序(1)中的语句序列不满足单赋值规则,经过修改之后,程序(2)中语句序列能够满足单赋值规则。单赋值规则使程序很清晰且易于理解,这比常规语言中重复使用同一个变量名所带来的好处更为重要。

单赋值规则是由 Tesler 和 Enea 于 1968 年首先提出的,这种规则为程序的并行执行提供了一种新方法。单赋值规则已经在法国的 LAU 数据流计算机和英国的 Manchester 数据流计算机中被采用。

(3) 不产生副作用

为了保证指令的执行顺序与数据相关性的约束条件相一致,任何一个程序在执行过程中必须具有不产生副作用的特点。在传统计算机的程序设计语言中,副作用主要来自如下几个方面:

- 使用了全局或公共变量;
- 不适当地扩大变量的使用范围;
- 使用了变量的同义名;

- 在子程序中修改了调用程序中的变量。

在数据流语言中,不使用全局变量和公共变量,严格控制变量的使用范围,采用赋值调用(call by value),而不是传统计算机中的引用调用,这就从根本上解决了变量的同义名问题。赋值调用过程只复制变量,而不修改变量。因此,在子程序中决不会修改调用程序传送来的变量。也就是说,为了避免产生不必要的副作用,各程序模块之间的输入和输出是完全隔离的。

(4) 结果的局部性

在传统的过程式语言(如 Pascal、Algol 和 C 等语言)中大量使用局部变量,这些变量只允许在本过程内部定义、赋值和引用,这就保证了操作结果具有局部性,不对其它过程产生影响,即使在不同过程中使用了相同名字的变量也仍然能够保证操作结果具有局部性。但是,在上述这些过程式语言中,也允许使用全局变量,并规定在任何一个过程中都可以对全局变量赋值,这种全局赋值可能对其它过程产生影响。在数据流计算机的指令中没有长远起作用的数据依赖关系,数据流语言完全采用模块化结构,不使用全局变量,不允许全局赋值,对形式参量的赋值也有严格的限制。因此,在数据流语言中每一个操作产生的结果都具有局部性。

(5) 循环程序迭代展开

如果一个数据流语言不能高效率地把循环程序展开进行迭代运算,那么它就不是一个好的数据流语言。从图 11.20 中可以看到,一个循环程序可以用一个环行结构的数据流程序图来表示,而数据流计算机本身就是一种环行结构。因此,数据流计算机对循环程序具有天然的适应性。但是,如果要把一个循环程序的不同次循环展开来进行并行计算,则必须在数据令牌中增加一种特殊的标记,这就是所谓动态数据流计算机的概念。下面举一个简单的例子来说明如何把一个循环程序展开进行并行计算。

例 11.4 一个用 C 语言编写的简单循环程序如下:

```
for(i=1;i<=n;i++)
{
    x[i]=a[i]+b[i];
    y[i]=x[i]+c[i];
    z[i]=x[i]+y[i];
}
```

在每个循环体中存在三个“先写后读”数据相关,因此这个循环程序的每个循环体只能串行执行。按照常规的计算方法,循环体部分只能用一个加法器运算,共需要 $3n$ 个机器周期才能执行完成。

如果把相邻的三个循环体同时展开,并按照流水线原理重新进行调度,可以得到语义上完全等效的新循环程序。

```
x[1]=a[1]+b[1];
y[1]=x[1]+c[1];x[2]=a[2]+b[2];
for(i=1;i<n-1;i++)
{
```

```

    z[i]=x[i]+y[i];
    y[i+1]=x[i+1]+c[i+1];
    x[i+2]=a[i+2]+b[i+2];
}
z[n-1]=x[n-1]+y[n-1];y[n]=x[n]+c[n];
z[n]=x[n]+y[n];

```

在经过重新调度之后的新循环程序中,循环体内的三个语句之间不再有任何数据相关,因此可以用三个加法器并行执行。仔细观察经过调度后的循环程序不难发现,新的循环体实际上跨越了原循环程序的三个相邻循环体。

循环程序前面的部分称为装入部分,循环程序后面的部分是排空部分,这两部分程序中同一行内的语句之间没有数据相关,它们可以并行执行。执行完成这个新循环程序只需 $n+2$ 个机器周期,运算速度提高三倍。

上面的方法与本书第四章中介绍的硬件流水技术非常相似,因此把这种方法称为软件流水 (software pipelining) 技术。在循环程序执行过程中采用软件流水技术能够把属于同一个循环程序的多个循环体重叠起来并行执行,能够使用多个处理机并行执行同一个程序,从而缩短程序的执行时间。

下面分别对几台典型的数据流计算机作简单的介绍,介绍它们的系统结构、机器组成、指令及数据令牌的格式等。

11.1.5 静态数据流计算机结构

按照对数据令牌的不同处理方法,数据流计算机可以分为静态数据流计算机和动态数据流计算机两类。静态数据流计算机的典型代表是 Dennis 计算机,而典型的动态数据流计算机除了 Arvind 计算机之外,还有英国的 Manchester 数据流计算机和日本研制的 EDDY 数据流计算机等。另外,在静态和动态两类典型的数据流计算机之外,近几年又出现了几台其它类型的数据流计算机。例如,美国犹他大学研制的树形结构的 DDM-1 数据流计算机,法国 Toulouse 等人研制的总线式 LAU 数据流计算机,英格兰人研制的数据流与控制流相结合的 New Castle 计算机及美国依利诺斯大学的 Kuch 和 Gajski 等人提出的同步与异步结合的 DSD-1 数据流计算机等。

11.1.5.1 静态数据流计算机的组成

Jack Dennis 是静态数据流计算机的先驱者,他及他的助手们于 1972 年首先提出了静态数据流模型,并根据所提出的模型研制了 Dennis 静态数据流计算机,在该机上运行了 VAL 数据流语言。

静态数据流计算机的主要特点是每个结点一次只能执行一个操作,反映到数据流程图图中,每条分支线上在同一时刻只能传送一个数据令牌。因此,静态数据流计算机的结点操作规则是:结点的每一条数据输入分支线上都有一个数据令牌出现,每一条控制输入分支线上都有控制令牌出现,而且这些控制令牌所携带的控制信号正好是这个结点操作所要求的,同时输出分支线上没有任何令牌,该结点的操作才能够被执行。所以,静态数据

流计算机中不仅要有数据令牌,还要有控制令牌,由这两种令牌同时来决定结点的操作是否执行。当数据令牌从分支线上被取走后,相应的控制令牌就作为回答信号(acknowledge signal)返回。

图 11.22 是 Dennis 静态数据流计算机的结构框图,它主要由五个部分组成。

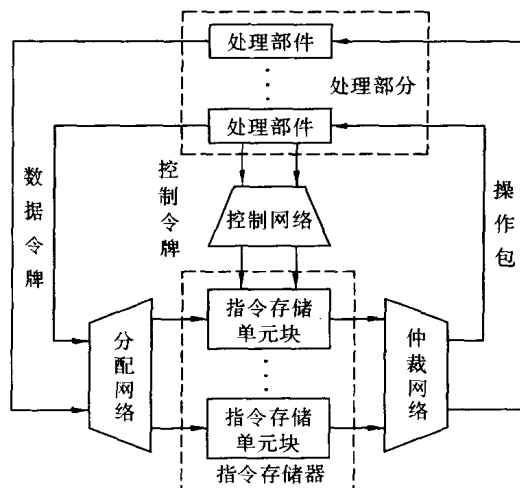


图 11.22 Dennis 静态数据流计算机结构框图

(1) **指令存储器**的作用是存放指令。每条指令有一个唯一的地址,也称为指令单元标识符。通常的一条指令由一个操作码、一个或几个操作数存储单元、一个或几个后继指令地址(目标地址)、一个或几个确认地址(回答信号地址)及有关标志等组成。其中,操作码唯一确定这条指令要执行的操作。操作数存储单元存放由数据令牌传送来的操作数,这些数据令牌是激活本条指令必需的。后继指令地址的作用是把本指令的执行结果送往需要它的那些指令中去。在本指令执行完后,由后继指令地址和结果数值一起组成新的数据令牌。

一条指令往往代表数据流程图中的一个或几个操作结点,还包括与这些操作结点有关的输入输出连接符。

(2) **处理部件**主要由多个相同或不同的处理单元组成。处理部件主要用来完成数据的函数运算。

(3) **仲裁网络**(arbitration network)的主要作用是把操作包(由指令及指令所要求的操作数组成)从指令存储器传送到处理部件。

(4) **控制网络**(control network)的主要作用是把控制令牌从处理部件传送到指令存储器。

(5) **分配网络**(distributed network)的主要作用是把数据令牌从处理部件传送到指令存储器。

存放在指令存储器中的指令从分配网络得到数据令牌中携带的操作数,从控制网络得到控制令牌,当指令所要求的所有操作数和控制令牌都到达之后就成为可执行指令。可执行指令从指令存储器中读出并和它所需要的操作数一起形成一个操作包,这个操作包

通过仲裁网络传送到处理部件。在处理部件中,指令执行后产生的结果组合成新的数据令牌和新的控制令牌,这两种令牌分别通过分配网络和控制网络送到指令存储器中。在指令存储器中这些令牌又可以激活其它指令。

每个指令单元中都有一个或几个接收缓冲器,这些缓冲器用来存放激活本指令的数据令牌中所携带的操作数。当一个指令单元接收到它所需要的全部操作数和全部控制令牌后,该指令单元中存放的指令就被允许执行,成为可执行指令。

仲裁网络在每个指令单元与每个处理部件之间都提供有通路,并根据操作码的初步译码结果在网络的各个输出端口之间分配操作包,一个处理部件在收到操作包之后,用操作包本身提供的操作数完成由指令规定的操作,并产生一个或几个结果令牌,结果令牌包括数据令牌和控制令牌两种。

根据数据令牌中后继指令地址部分的指示,分配网络把数据令牌中所携带的操作数送入相应指令单元的接收缓冲器中。也就是说,数据令牌实际上是按照令牌本身所携带的后继指令地址传送的。当一个数据令牌到达它所要求的指令单元时便为该指令单元提供了一个操作数。

处理部件由功能相同或不同的多个处理单元组成,这些处理单元能够并行地处理多条指令。指令的执行由控制网络控制,其执行顺序仅受数据相关性的约束。指令执行完成后产生新的数据令牌,而令牌中的后继指令地址部分直接从被处理的指令中取得。因此,在数据流计算机中指令执行结果不像传统计算机中那样送入共享内存单元或共享通用寄存器中,而是通过数据令牌直接传送到需要它的指令单元中去。这样,指令在执行时就不必到内存或通用寄存器中去取操作数了。

Dennis 数据流计算机中的五个组成部分相互独立,没有统一的中央时钟,各部件之间采用异步方式互相通信。所有需要传送的信息都组成有关信息包,通过信息包在各子系统之间传送操作数和控制信号。

由于在同一时刻可能有多条指令在等待执行,因此仲裁网络应设计成允许多个操作包同时通过。同样,分配网络和控制网络也要设计成能把密集的令牌同时分配到有关指令单元中去。通常一台数据流计算机中的处理单元有几十个到几千个,而指令存储单元的数目就更多了。因此,通信网络往往是数据流计算机工作时的瓶颈。换句话说,如何设计好三个通信网络就成为设计整个数据流计算机的关键。

11.1.5.2 分块结构的静态数据流计算机

当指令单元的数目很多时,可采用如图 11.23 那样的分块结构。把指令单元、分配网络和仲裁网络都分成若干个单元块,在单元块内部实行共享连接,而单元块之间采用交换网络连接。

图 11.23 中的分配网络、仲裁网络及交换开关可以采用如图 11.24 所示的基本单元组成,其中分配器是无阻塞的。仲裁器中的两个输入在同一时刻只能有一个被连接到它的输出。 2×2 交换开关共有 9 种可能的组合状态,其中有两种状态可能引起阻塞。 3×2 交换开关共有 27 种可能的组合状态,其中有 14 种状态可能引起阻塞。当阻塞发生时,在所有发生冲突的请求中只能有一个请求通过交换开关。如果网络中没有设置缓冲器,被阻塞

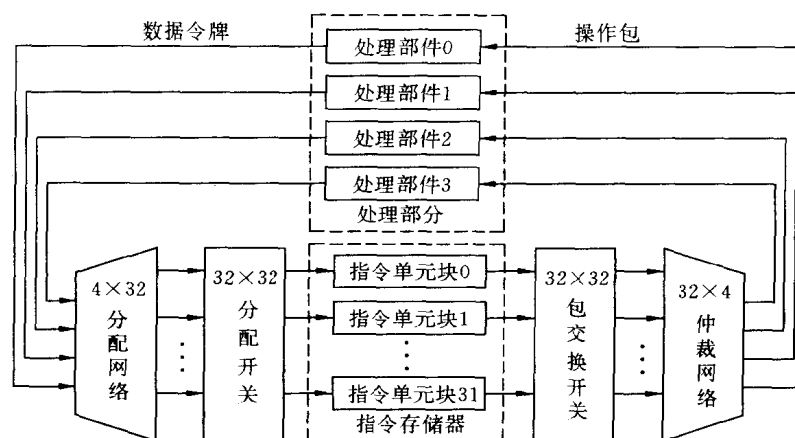


图 11.23 一种分块结构的数据流计算机模型
(由 4 个处理单元和 32 个指令单元块组成)

的请求将被丢失,只能在以后重新提出请求。对于在开关和仲裁器的输入端设置有缓冲器的包交换网络,其被阻塞的请求将保存在缓冲器中,以后再向输出端传送。

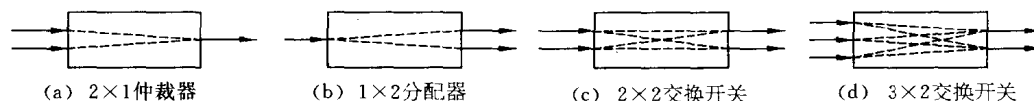


图 11.24 在数据流计算机通信网络中使用的仲裁器、分配器和交换开关

图 11.25 给出一个 27×8 有缓冲器的仲裁网络实例。图中每个 3×2 开关的每个输入端都有一个缓冲器。采用循环轮换法可以解决指向同一个输出端口的多个请求之间的冲突问题。通常在控制网络中不设置缓冲器,只通过直通的组合逻辑线路传送信息。

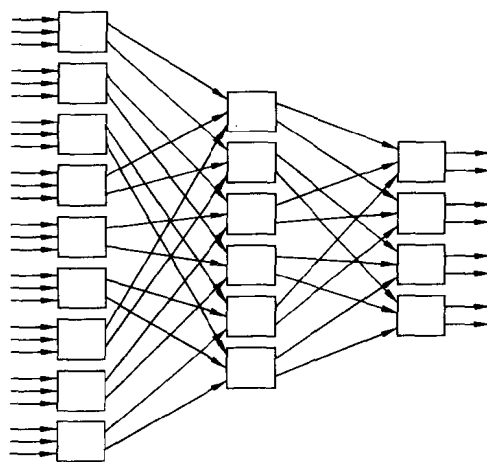


图 11.25 用 3×2 仲裁器组成的 27×8 带有输入缓冲器的仲裁网络

为了减少仲裁网络、分配网络和包交换开关中仲裁器、分配器及交换开关的数量,在

分配网络、指令单元(或指令单元块)及仲裁网络之间采用串行方式传送信息是可取的。为此,在信息包经过仲裁网络时要经过串行到并行的转换,而在分配网络中又要进行并行到串行的转换。然而,当需要传送的信息包流量很大时必须采用并行方式传送。

11.1.5.3 多处理机结构的静态数据流计算机

可以采用多个独立的数据流处理单元通过通信网络连接成一种多处理机结构的静态数据流计算机。图 11.26 是美国 Texas 仪器公司研制的并行静态数据流计算机。它由 4 个独立的静态数据流处理单元组成。

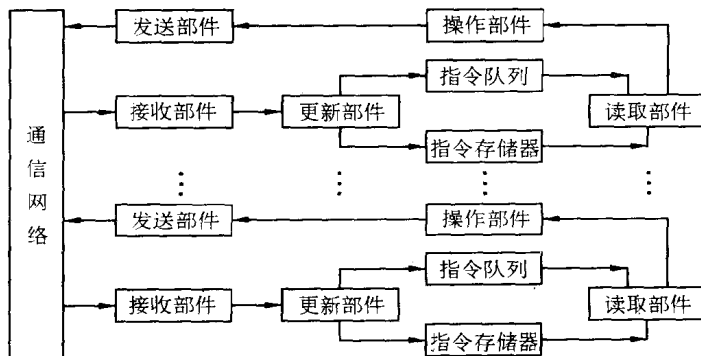


图 11.26 Texas 仪器公司设计的静态数据流计算机结构
(由 4 个独立的数据流处理单元组成)

为了能够对数组等结构型数据进行快速处理,并能够有效地压缩存储空间,在有的数据流计算机中增加了结构存储器和结构处理部件。由结构处理机和基本数据流处理机共同组成一种多处理机结构的数据流计算机,如图 11.27 所示。

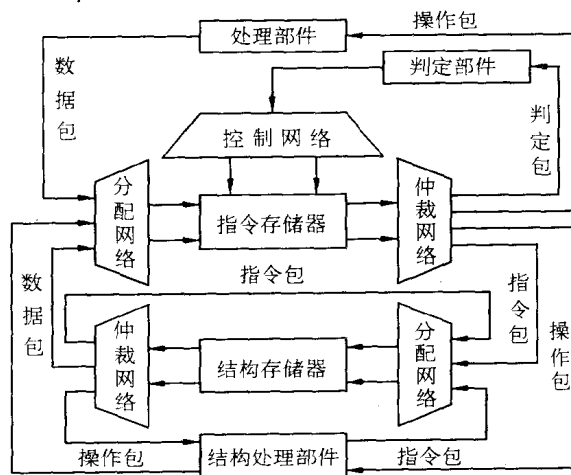


图 11.27 具有结构存储器和结构处理部件的数据流计算机

图中下半部分为结构型数据处理机,包括结构存储器、结构处理部件及相应的仲裁网

络、分配网络等。由基本数据流计算机(上半部分)中的仲裁网络作出判断,把那些要求建立新的结构操作结点的指令送到结构处理部件和结构存储器中去执行。而那些可直接执行的指令仍在常规的数据处理部件中执行。

11.1.5.4 静态数据流计算机作为后端机

目前,许多静态数据流计算机仅仅作为主机(host)的一个后端机来使用,而主机仍然采用传统的冯·诺依曼型计算机。数据流计算机指令存储器中的数据流程图由主机(传统的冯·诺依曼型计算机)加载而来。

图 11.28 是一台已经用于气象模型计算的 Mondala 静态数据流计算机。其中, M 是指令存储器,它由 n 个存储模块组成,每个存储模块又分为许多个存储单元块,每个存储单元块存放数据流程图中的一个操作结点。另外,为了支持快速和高效的数组访问,存储器中的一部分用来存放数组。PE 为处理单元,共有 m 个,这些处理单元能处理 32 位的浮点操作、定点操作、逻辑操作、移位操作和通信操作等。

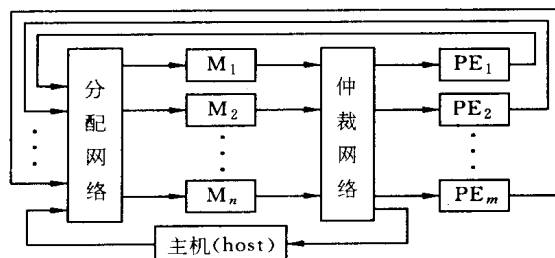
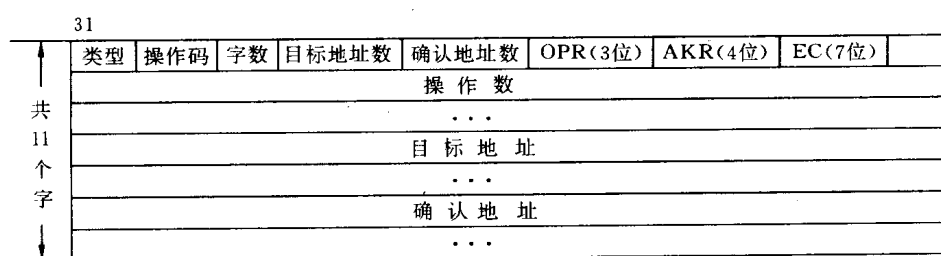
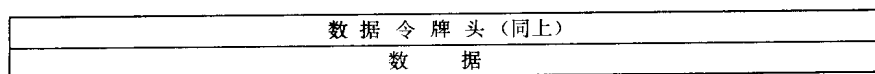


图 11.28 Mondala 静态数据流计算机结构框图

在后端机运行过程中,指令存储器 M 中的可执行指令通过仲裁网络传送到处理单元 PE,通过分配网络把指令执行后产生的数据令牌中的数据送入有关存储单元块中。指令和数据令牌的格式如图 11.29 所示。



(a) 指令格式



(b) 数据令牌格式

图 11.29 Mondala 数据流计算机的指令和数据令牌格式

数据流程图中的每个结点各用一条指令来表示,它共占用 11 个 32 位的指令存储

空间(一个存储单元块)。在指令头(指令的第一个 32 位字)中,OPR 字段记录本指令需要用到的操作数个数,共 3 位。AKR 字段记录本指令需要收到的确认信号(控制信号)个数,共 4 位。EC 为本指令的可执行计数器。EC 对收到的数据令牌个数和控制令牌个数分别计数,当 EC 中的内容与 OPR 和 AKR 连接在一起(共 7 位)的寄存器中内容相等时,对应存储单元块中存放的指令即成为可执行指令。

在指令处理单元中不设置指令和操作数寄存器。当同时有几条可执行指令被分配到同一个处理单元时,只有一条指令能在处理单元中执行,其余可执行指令以先进先出方式暂存于仲裁网络的缓冲存储器中。处理单元的输出端也设置有缓冲存储器,这个缓冲存储器通过分配网络与指令存储器相连接。

Mondala 静态数据流计算机还支持向量指令和复杂的多重操作指令。这样做虽然处理部件要复杂些,但将大大减少可执行指令和数据令牌等在系统中的传送开销。另外,Mondala 计算机还支持虚拟页式存储器管理,具有多用户环境。

11.1.6 动态数据流计算机结构

在动态数据流计算机中,每个数据令牌都带有标志(令牌标号及其它特征信息),从而使数据流程图中的一条有向分支线上可同时传送几个数据令牌,而且不需要用控制令牌来确认指令和数据令牌的传送。它采用一个专门硬件(匹配部件)对数据令牌中的标志进行符合比较,以便对需要多个操作数的指令进行数据令牌的合并。当一条指令所要求的数据令牌都到齐后,就立即从指令存储器中取出这条指令,并把该指令与数据令牌中携带的操作数一起组成一个操作包送入可执行指令队列。如果指令所要求的数据令牌没有全部到齐(匹配失败),则把刚到达的数据令牌暂时存入匹配部件的缓冲存储器中,以供下次匹配时再使用。匹配缓冲存储器通常是一个相联存储器。

从原理上分析,动态数据流计算机能更加充分地开发程序中的并行性,且中间结果不返回存储器,从而减少了操作开销。

动态数据流计算机从结构上分为三大类。第一类是以 Arvind 为代表的网络结构型计算机,第二类是以 Manchester 为代表的环形结构计算机,第三类是以 EDDY 为代表的网状结构数据流计算机。下面介绍这三类动态数据流计算机的结构特点、工作原理、指令及数据令牌的格式等。

11.1.6.1 网络结构的动态数据流计算机

Arvind 动态数据流计算机的研究最初是在加州大学的 Irvine 分校开始的,后来移到

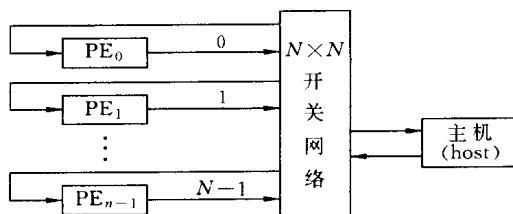


图 11.30 Arvind 动态数据流计算机结构

MIT 科学实验室由 Arvind 及其助手们继续进行,并彻底修改了原来的设计方案。现在的 Arvind 动态数据流计算机结构如图 11.30 所示。它不采用令牌环结构,而是用 $N \times N$ 的开关网络进行 PE 之间的通信,构成一种网络结构的动态数据流计算机。 N 个 PE 与 $N \times$

N 个开关网络之间采用按位串联方式交换信息。

图 11.30 中的每个 PE 都是一台完整的数据流处理机。在图 11.31 中给出一个 PE 的结构框图。图中输入部分有一个缓冲寄存器用来存放从开关网络送来或从本 PE 的输出部分送来的数据令牌,并对该令牌进行初步译码。如果还需要另一个数据令牌与之匹配,那么首先在匹配缓冲存储器中寻找。如果需要的那个令牌还没有到达,则把本令牌暂时存入匹配缓冲存储器中。当匹配所需要的数据令牌都到齐后,便把这些令牌组成一个令牌组一起送到指令读取部件中去。指令读取部件按照令牌中给出的本条指令地址从本地指令存储器中读出这条指令。读出的指令立即送入可执行指令队列,一旦算术逻辑部件 (ALU) 有空闲就立即执行这条指令。

I 结构 (I structure) 存储器用来存放结构型数据,如数组等。采用 I 结构存储器可避免大量复制数据。这种存储器中的每个元素都有一个有效位,如果要读取 I 结构存储器中的一个元素,则要首先判断这个单元中的有效位是否置位,如果没有置位则本次读取要延迟。

当算术逻辑部件及目标读取部件执行完成后,把目标 PE 号及有关标志一起组成结果令牌送到输出部分的缓冲器中,这样就完成了一条指令的执行过程。送到输出缓冲器中的结果令牌将作为新的数据令牌来使用。

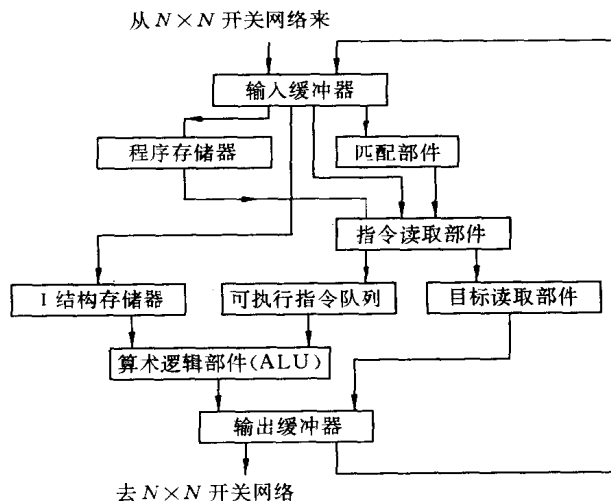


图 11.31 Arvind 动态数据流计算机中一个 PE 的结构框图

在数据流计算机执行一个过程(或一个循环体)时,通常要为其分配一组 PE,这个过程的所有操作都在这组 PE 内执行。如果在一个过程中又有多个并行代码块,则要对每个块赋予不同的“颜色”。当一个 PE 组中的所有“颜色”都用完后,就不再向这个 PE 组中分配新的过程。在代码块或过程执行完成后则释放“颜色”。采用“颜色”控制有利于共享代码块,因为所有活动着的代码块都有不同的“颜色”,从而增加了资源的利用率,也提高了整个系统的吞吐率。

图 11.32 给出 Arvind 动态数据流计算机典型的指令格式及数据令牌格式。其中 OP

是操作码,AM 是操作数寻址方式,nc 表示本指令中包含的常数个数(最多 2 个),nd 为结果令牌的目标个数。每个目标由 4 个部分组成,其中 S 是目标指令地址,p 是目标指令的输入端口,nt 表示为激活目标指令所需要的数据令牌个数,of 是当目标指令执行时 PE 要完成的功能。

OP(8 位)	AM(3 位)	nc(1 位)	nd(1 位)
C1 (常数 1)			
C2 (常数 2)			
S(16 位)	p(1 位)	nt(1 位)	of(1 位)
S(16 位)	p(1 位)	nt(1 位)	of(1 位)
S(16 位)	p(1 位)	nt(1 位)	of(1 位)

(a) 典型指令格式

处理机号	类型	颜色	本地指令地址	迭代数	令牌数	端口
数据 (32 位)						

(b) 数据令牌格式

图 11.32 Arvind 动态数据流计算机的指令及数据令牌格式

Arvind 动态数据流计算机采用 ID 语言。ID 是一种高级数据流语言编译器,它可以把用户用高级语言编写的源程序直接编译成数据流程图。编译工作在主机上进行,编译好的数据流程图通过 $N \times N$ 开关网络加载到有关 PE 上。

11.1.6.2 环形结构的动态数据流计算机

Manchester 动态数据流计算机是环型结构计算机的典型代表,其结构框图如图 11.33 所示,它的五个功能部件按顺时针方向进行通信,组成一条环形流水线。数据令牌是主要的通信单位,令牌主要由操作数、标号及目标结点指针等部分组成。

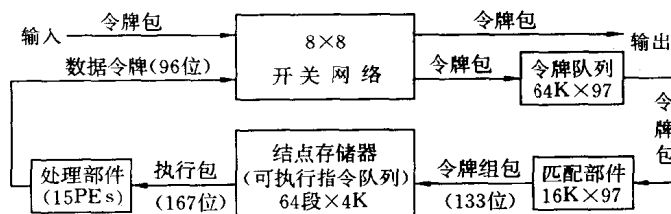


图 11.33 Manchester 动态数据流计算机结构框图

处理部件由 15 个 PE 组成,这些 PE 可并行地执行指令,它们可执行定点、浮点、数据转移及打标记等指令。每个 PE 都有输入缓冲器和输出缓冲器。

8×8 开关网络可同时提供多条通路与外部交换信息。令牌队列可存放 64K 个数据令牌。匹配部件按照令牌的特征值对令牌进行匹配,在它的内部有 $16K \times 97$ 位的缓冲存储

器。缓冲存储器由 8 组相联存储器组成,采用硬件散列技术来减少相联比较器的位数。当从令牌队列中送来的数据令牌与匹配部件中已经存在的令牌相匹配(有相应的特征值)时,表示令牌中目标地址字段指示的指令为可执行指令,于是 97 位的数据令牌和 36 位的匹配特征值合在一起组成 133 位的令牌组包送往结点存储器。如果从令牌队列送来的数据令牌不能与匹配部件中已经存放的令牌相匹配时,则把新送来的令牌暂时存入匹配部件的缓冲存储器中。

结点存储器按照匹配部件送来的令牌组包中给定的目标地址取出指令,并把令牌组包中携带的操作数代入指令中,形成 167 位的执行包送往处理部件。

数据令牌与指令格式如图 11.34 所示。

系统/计数标志	1 位
特征值(tag)	36 位
目标地址	32 位
数 值	37 位

(a) 数据令牌格式

系统/计数标志	1 位
特征(tag)	36 位
操 作 码	12 位
操 作 数 1	37 位
操 作 数 2	37 位
目标地址 1	22 位
目标地址 2	22 位

(b) 指令格式

图 11.34 Manchester 动态数据流计算机的指令及数据令牌格式

Manchester 动态数据流计算机采用高级数据流语言 Lapse 编程,这是一个单赋值的程序设计语言,其中的语法规则与 Pascal 语言基本相同。Lapse 语言还可用一个语句描述整个数组的运算。

11.1.6.3 网状结构的动态数据流计算机

日本研制的用于科学计算的动态数据流计算机 EDDY(experimental system for data driven processor array)如图 11.35 所示,它主要由一个 4×4 的二维网状阵列共 16 个 PE 及两个播送控制部件(BCU)组成。每个 PE 能够与其相邻的 8 个 PE 直接相连接。播送控制部件可以同时按行或按列把程序和数据装入所有 PE 或从 PE 取走。在最初的设计中,每个 PE 由两个微处理器 Z8001 组成,并在该系统上做了详细的环形数据流流水线模拟,取得了大量的数据。根据这些模拟结果,后来专门为 EDDY 数据流计算机设计了新的 PE 硬件。

新的 PE 采用环形流水线结构,如图 11.36 所示。它由指令存储器 IM、操作数存储器 OM、操作部件 OU 及通信部件 CU 四大部分组成。当一个数据令牌到达指令存储器时,指令存储器就取出这个令牌指示的指令,并把它与数据令牌中携带的操作数一起送到操作数存储器。如果到达的数据令牌只是双操作数运算中的一个数据令牌,则操作数存储器采用相联访问方式寻找与这个数据令牌相匹配的另一个操作数,如果找到这个配对的操

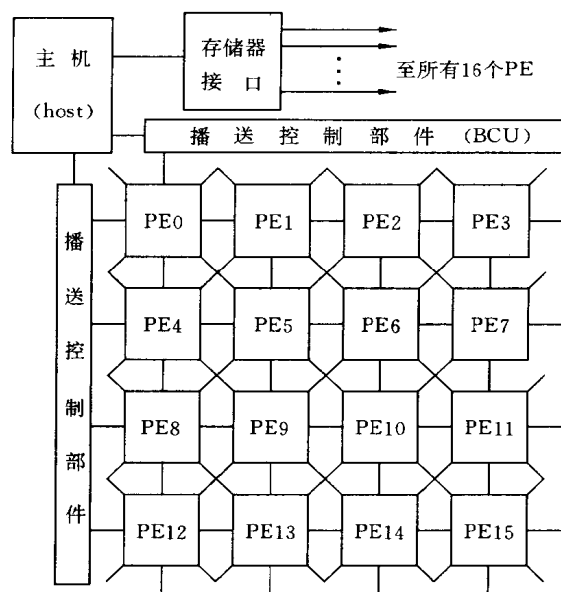


图 11.35 EDDY 动态数据流计算机结构

作数,就组成一个操作包送到功能操作部件中去执行。如果找不到配对的操作数,则把刚到达的数据令牌暂存入操作数存储器中,并附加上一个关键字。

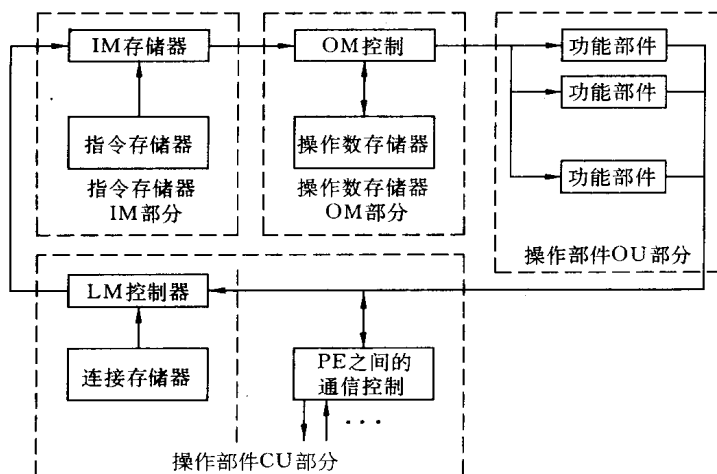


图 11.36 EDDY 动态数据流计算机中的一个 PE

通信部件由两个部分组成,一个是连接存储器,另一个是 PE 间的通信控制器。其中,通信控制器的作用是把结果包发送给连接存储器或相邻的 PE,当然也可以从其它 PE 接收结果包送入自己的连接存储器。

EDDY 数据流计算机采用函数语言 VAL 及 ID 作为它的程序设计语言,这两种语言都是典型的单赋值语言。

11.1.7 其它类型的数据流计算机

在上述静态和动态两种典型的数据流计算机之外,还有几种其它类型的数据流计算机。其中有的克服了典型数据流计算机某些方面的缺点,有的继承了传统计算机中已经证明行之有效的某些并行处理技术。因此,从某种意义上说,这些结构类型代表了目前数据流计算机的发展方向。归纳起来,有以下五种类型。

11.1.7.1 利用传统多处理机结构的数据流计算机

美国第一台实际可运行的数据流计算机 DDM-1(data device machine-1)采用树型结构。其中的每个单元可以连到一个祖先单元和最多 8 个子单元。DDM-1 是由犹他大学的 A. Davis 及其同事们在 1976 年设计的,其程序及机器结构都建立在递归的概念上。它以 DEC 20/40 作主机,而 DDM-1 仅仅作为主机的一个后端机使用。程序的编辑、编译、模拟及机器性能的测试等均在主机上进行。

法国 Toulouse 设计的 LAU 数据流计算机系统通过多总线互连,由 32 个位片微处理器组成。其程序设计语言采用单赋值规则,但它的机器组织仍采用传统的控制流概念,通过共享的存储单元传送数据。数据的访问通过指令中给出的地址进行。另外,它采用专门的控制信号来激活指令。LAU 数据流系统于 1983 年投入实际运行。

11.1.7.2 提高并行级别的数据流计算机

普遍认为限制数据流计算机发展的主要原因之一是机器的操作开销太大。而操作开销大的根本原因是数据流计算机的设计者们把并行主要放在指令级上,由此被称为数据驱动计算机。实际上,对于一个计算机系统,并行可划分为多个级别,从高到低主要有:任务级并行、作业级并行、进程级并行、过程级(复合函数级)并行、函数级并行、指令级并行等。而在一个程序模块内往往存在多种并行性,如果把高一级的并行性都依赖于指令级的并行性来实现,仅仅借助尽可能多的指令同时执行来挖掘各级的并行性,这种做法将会付出过高的代价。例如,数组型数据结构的处理就是一例。如果把并行提高到函数一级或复合函数一级,利用数据直接驱动函数、复合函数、进程或作业运行,这样做付出的总的操作开销就小多了。

Gajks 等人于 1982 年和 Motooka 等人于 1981 年各自分别提出一种复合函数(compound function)级驱动方案,也称相关驱动(dependence driven)方案,它把并行放在复合函数一级,从而简化了控制,减少了操作开销。伊利诺伊大学的科研小组为此研制了六种复合函数,它们是:数组(向量、矩阵)运算、线性递归、全操作(for all)循环、流水线循环、赋值语句块及复合条件语句。

在复合函数驱动方案中,数据流程图用有向分支线直接连接复合函数结点(而不是操作结点)构成,并且可以用传统的高级语言来编写程序,这样可以继承长期积累起来的大量软件。

为了把传统高级语言编写的程序转换成数据流程图,并从数据流程图生成代码,需要另外研制专门的程序转换软件。图 11.37 表示复合函数驱动方案与传统的数据驱动

方案从高级语言到代码生成的编译过程。从图 11.37(a)与图 11.37(b)的比较中看出,复合函数驱动方案要额外增加两个转换模块,其中一个模块把传统高级语言编写的程序转换成由复合函数结点连接而成的数据流程图,另一个模块是要把数据流程图中的有向分支线和结点转换成目标代码。

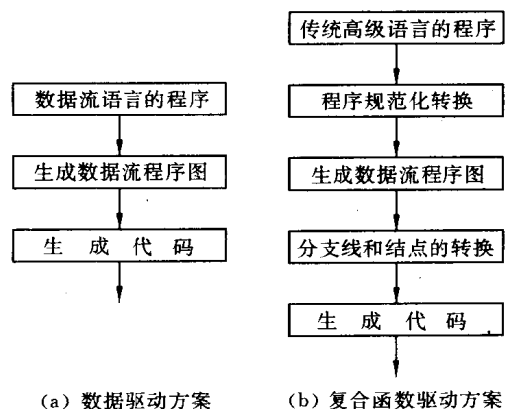


图 11.37 复合函数驱动与数据驱动两种方案中高级语言编译过程的比较

11.1.7.3 采用多级并行的数据流计算机

把上面讲述的复合函数驱动方案进一步推广,可以在一台计算机中同时采用任务级、作业级、进程级、复合函数级、函数级及指令级多级驱动方案。这种方案在 1983 年由

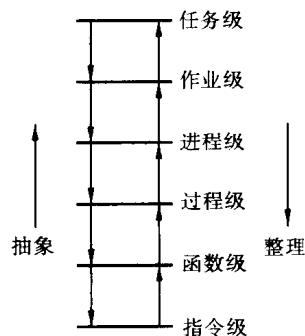


图 11.38 在同时采用多级并行性的数据流计算机中程序的抽象与整理示意图

Hwang 和 Su 首先提出,它把一个程序经过适当的抽象(abstraction)或整理(engrossment)后在多个并行级别上加以定义,如图 11.38 所示。所谓抽象是指从最低级(指令级)到最高级(任务级)挖掘程序中的并行性,而整理则正好相反。通过这两种方法使程序中的并行性得到充分开发,同时也大大减少操作开销。

在同时采用多级并行的数据流计算机中,关键技术是研制把一个程序进行抽象或整理的机构,它需要进行多层次的调度,而且又不能化费太大的系统开销。通常可以分两部分来实现,其中一部分在编译时完成,另一部分在运行时完成。这二者如何选择取决于在多个不同层次上促进并行处理所使用的性能准则。研究如何进行层次式资源调度是这个方法的最关键部分。

经过抽象或整理后,一个程序被分解为从任务级、作业级、进程级、复合函数级、函数级及指令级多种级别并存的一个个“事件”。接下去的问题是如何把这些事件分配给可用的资源。在数据驱动计算机中,一般对可执行指令采用先进先出的调度策略,而在同时采用多级并行的数据流计算机中则不同,通常要采用优先队列(priority queues)来进行调度。优先级是在程序运行前估计所有可运行事件的时间/空间复杂性而确定的。

由逻辑事件向物理资源的最佳映射,已被证明是 NP 完全类(NP complete)问题。如果对复杂性的估计足够的话,用优先队列的试探法可以得到接近最佳的性能。从直观上来说,这种多层次并行方案更有助于设计出既采用数据流又采用控制流的通用计算机。但是,由于层次控制的复杂性,为了使这种方案切实可行并具有良好的性能价格比,还需要做大量的研究开发工作。

11.1.7.4 同步与异步相结合的数据流计算机

有一种看法认为数据流计算机操作开销太大的基本原因之一是完全采用异步操作而没有集中控制。异步操作造成开销大的原因有如下三个:

(1) 在任何具体的数据流计算机中,因为物理资源都是有限的,所以各部件都要设置缓冲存储器,用以暂存等待服务的信息,还要设置排队部件,并按照某种服务策略选择服务对象。

(2) 在数据流计算机内部,数据令牌、控制令牌、可执行指令及因为异步操作而产生的各种回答信号等在系统内异步、随机地传送,为了克服由此引起的混乱状态需要设置复杂的通信及控制机构。

(3) 由于采用异步操作,数组等结构型数据只能分解为一个个孤立的元素参加运算,而不能采用已经相当成熟的向量流水技术。

实际上,异步操作与前面提到的并行性一样也可分为多个层次,如指令级异步操作、函数级异步操作等。虽然异步操作有许多不容置疑的优点,但是不适当地在各级都采用异步操作,特别是在指令级采用异步操作是造成数据流计算机操作开销很大的基本原因之一。因此,有一种看法认为,在指令级采用同步操作,而在函数级及函数以上级采用异步操作是克服数据流计算机操作开销过大的有效途径。

国防科技大学计算机系设计的 SDS-1 数据流计算机如图 11.39 所示,它由四个标量处理部件 SPU0~SPU3、四个向量处理部件 VPU0~VPU3、一个程序及标量数据存储器 SM、一个数组存储器 AM 及一个全局控制器 GCU 等组成。标量数据、程序和函数标题都存放在 SM 中,数组存放在 AM 中。在函数标题中登记有函数的程序地址、输入数据计数器及运算结果的函数标题地址等,而向量数据则存放相应的指针。

SDS-1 在低一级实现指令并行同步操作,而在高一级实现函数并行异步操作。因为在指令级采用同步操作,中间结果不返回存储器,直接进入下一轮操作,因此在指令中无须携带目标地址,实际指令长度仅 21 位。由于同步操作不需要回答信号,系统的通信量很小,整个系统采用总线结构已完全能够胜任。在函数级采用并行异步操作虽然开销大,如取函数标题、取程序等占用时间长,函数标题也占用了存储空间,但这些开销平均到函数中的每条指令上也就所剩无几了。

11.1.7.5 控制流与数据流相结合的数据流计算机

数据流计算机的研究人员普遍认为,在小规模并行系统中,数据流原理已被证明是行之有效的,而且已经得到了成功的演示。当同时性较低、无规则且与运行相关时,数据流计算机是应该选择的系统结构。另一方面,在大规模并行系统中,由于操作开销大的问题被

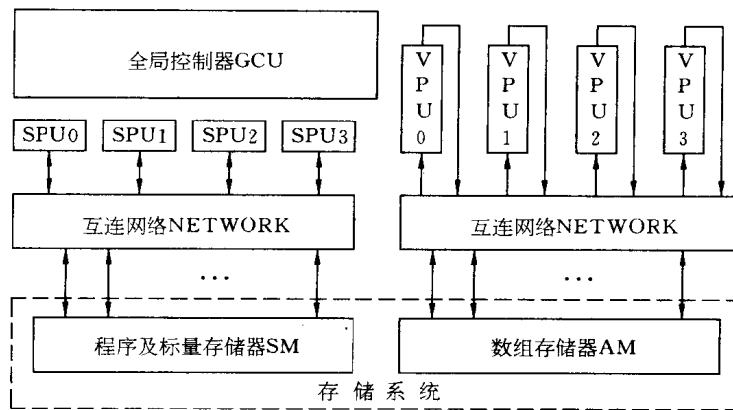


图 11.39 SDS-1 同步与异步相结合的数据流计算机框图

很高的并行度所掩盖,因此也有较好的发展前景。但在中等规模的并行系统中,数据流计算机很少有希望获得成功,在这个领域中,流水线处理机、向量处理机及传统的多处理机更为有效。另外,也有许多人对数据流语言的前景提出异议。至今,高速计算机的市场已经被保守性和软件兼容性占领。目前这种状况下的数据程序设计语言能战胜这种保守性吗?

美国伊利诺伊大学的 Gajsk 和 Kuck 等人对数据流原理的优缺点作了充分的分析,批评它固有的低效性,提出自己的发展途径。他们认为,要继承传统控制流计算机的优点,采用控制流与数据流相结合的方法来发展数据流计算机。他们设计的 Cedar 数据流计算机系统把并行级别确定在函数级,重点实现复合函数级的并行操作,称为宏流水线。而在指令级仍采用传统的控制流方法实现。另外,还采用了控制流计算机中的向量处理技术。在软件方面,选用传统的 Fortran 语言,由编译器来挖掘程序中的并行成分。

Cedar 数据流计算机系统既有数据流技术的优点,实现函数级并行操作,使控制简单,操作开销也不大,又有传统控制流计算机中已经被证明行之有效的技术,并使用传统

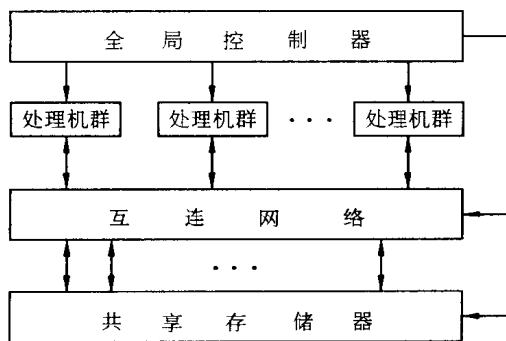


图 11.40 建议在高级数据流计算机系统中采用的硬件结构

高级语言编程,实现软件继承。因此它代表了当前数据流计算机研究中的一个值得引起人们注意的动向。

最后,给出一个建议在高级数据流计算机中采用的硬件结构,如图 11.40 所示。从这个结构出发,既可以设计出同步与异步相结合及同时采用多种并行级别的数据流计算机系统,也能设计出控制流与数据流相结合的计算机系统。

在图 11.40 所示的硬件结构中,有一个全局控制器,用来控制多个处理机群、共享存储器和其它资源,而不像典型的数据流计算机那样强调分散控制。这个结构与传统的多处理机结构非常相似,所不同的只是在某一个或某几个级别上采用数据驱动。因此,

这种数据流计算机很有可能成为将来的发展方向。

11.2 数据库机与知识库机

目前世界上正在运行中的绝大多数计算机都属于冯·诺依曼系统结构,这种结构的计算机主要是为解决复杂而费时的数值计算任务而设计的。但是,随着计算机应用领域的不断扩大,当前计算机解决的非数值处理任务已经超过了数值计算任务。造成这种情况的主要原因是人们在现实生活中遇到的问题大多数是非数值问题。例如,企业管理几乎都是符号处理和符号推理,所有的法律问题,大部分的生物、医学问题,甚至物理、化学,以至于数学等这些基础学科,其主要问题也是推理,而不是计算。

如果采用传统的冯·诺依曼结构的计算机来解决非数值任务,很明显的一个问题是速度很慢、效率很低。为了提高计算机处理非数值问题的速度,人们提出了许多解决办法。例如并行操作、相联查询(按内容查询)等。虽然也取得了一些成果,但是这些研究工作都没有抓住非数值任务的关键,因而也就没有突破性的进展。

非数值问题的焦点是知识存储(knowledge storage) and 知识处理(knowledge processing)问题。计算机能够在上述各个领域里得到广泛应用,最主要的一个原因就是数据库(DB:data base)和知识库(KB:knowledge base)的出现并获得广泛应用。特别是近几年来蓬勃发展起来的数据库管理系统(DBMS:data base management system)和知识库管理系统(KBMS:knowledge base management system)能够满足多方面不同用户的需要,为用户提供强大的知识查询语言(KQL:knowledge query language)和知识操纵语言(KML:knowledge manipulation language)。

然而,当数据库和知识库的应用越来越广泛的时候,人们发现冯·诺依曼式的单一处理机结构对知识的一致性保证和快速实时响应都是巨大的障碍。虽然采用并行处理和相联查询等方法能够提高响应的速度,但是并没有解决现代数据库管理系统和知识库管理系统的键问题。于是,这方面的研究者们提出了一个新名词——数据库机(DBM:data base machine)和知识库机(KBM:knowledge base machine),试图以此来解决访问数据库和知识库时的长时间等待,同时也增加处理数据和知识的灵活性。在数据库机或知识库机中,原来全部由软件实现的数据库管理功能和知识库管理功能部分或全部改由面向应用的硬件来实现。

数据库机和知识库机能够实现数据和知识的采集、存储、查询、删除和修改等功能,能够有效维护数据库和知识库的一致性及其完整性。

11.2.1 数据库机与知识库机模型

究竟什么是数据库机或知识库机?这个问题一直是研究者们十分关心的问题。根据目前的研究成果及数据库机和知识库机在整个计算机系统中的作用,可以这样来定义数据库机和知识库机:数据库机或知识库机的基本概念是从主计算机中移出一部分或全部数据库或知识库管理系统的功能,而把这些功能装入到一个后端系统中。原则上,这个后端系统可以是一台通用计算机,也可以是一台专门用于数据管理或知识处理的专用计算

机。由于这种方案可能只移出了一部分数据库管理系统或知识库管理系统的功能,因此,通常认为数据库机或知识库机仅仅是一个子系统,或者称为后端机(backend),而不是一台完整的计算机系统。

在通常的计算机系统中,数据库管理系统或知识库管理系统及与它们有关的联机输入输出部件(on-line I/O)等都装在同一台主机(host)上,而且主机就采用传统的冯·诺依曼计算机,如图 11.41 所示。

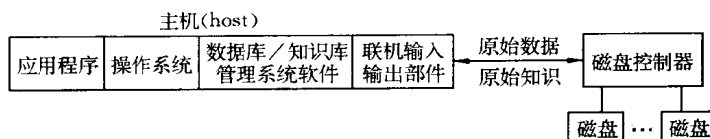


图 11.41 传统计算机上的数据库和知识库管理系统

图 11.41 是目前使用最广泛的数据库或知识库管理系统,它在一台通用计算机中装入一个数据库管理系统软件或知识库管理系统软件。在这种系统中,从磁盘存储器中读出的是未经加工的原始数据和原始知识。数据和知识的一切处理工作都在这一台传统的处理机上通过数据库管理系统软件或知识库管理系统软件进行。

而数据库机和知识库机则不是这样的,它们采用专门的后端机把数据库管理功能或知识库管理功能,包括联机输入输出功能从传统的主机中独立出来,并且采用特殊的硬件和软件实现这些功能。

根据各种后端机实现方法的不同可以有多种不同类型的数据库机和知识库机。例如,软件后端机(software backend)数据库机与知识库机、智能控制后端机(intelligent controller backend)数据库机与知识库机、硬件后端机(hardware backend)数据库机与知识库机等

11.2.1.1 软件后端机数据库机与知识库机

软件后端机数据库机与知识库机的模型如图 11.42 所示,它不需要专门设计任何特殊的硬件。通常可以采用一台传统的计算机作后端机,由这台后端机用软件完成数据库和知识库管理系统的功能,完成联机输入输出功能。后端机不直接面向用户,而由主机提供用户界面。当用户需要使用数据库机或知识库机时,只要发出有关请求信息即可。数据和知识的查询、操纵等功能都在后端机上完成,完成后的结果送回主机。

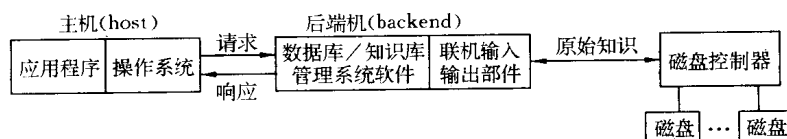


图 11.42 软件后端机的数据库机与知识库机模型

另外,后端机也可以是一台多处理机系统,如图 11.43 所示。其中的每一台后端机都可以采用传统的冯·诺依曼结构,只要把同样的软件装入到每一个后端机中,并采用一个

后端控制器来协调各个后端机的工作。

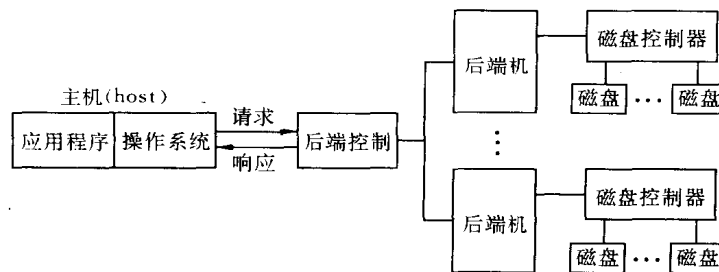


图 11.43 多个后端机的数据库机与知识库机模型

11.2.1.2 智能控制后端机数据库机与知识库机

智能控制后端机数据库机与知识库机的模型如图 11.44 所示,它需要采用专门设计的硬件作为后端机。通常的后端机由多个专门设计处理机、采用相联方式访问存储器和通信部件等组成。

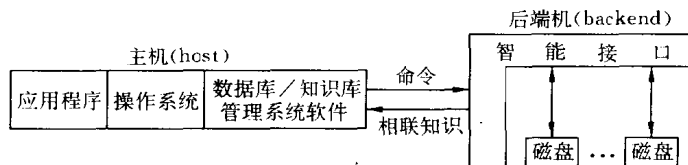


图 11.44 智能控制后端机的数据库机与知识库机模型

在智能控制数据库机与知识库机中,数据和知识的管理功能仍然在主机上用软件实现。当主机执行到数据或知识的操纵功能时,只要向后端机发出一个相应的命令,实际的数据和知识操纵功能在后端机上用硬件完成。后端机执行完成后再把结果送回主机。因此,它与软件后端机的数据库机或知识库机不同,它发给接口部件的不是简单的、访问原始知识的请求,而是一些比较高级的数据和知识处理命令。智能接口能够对原始数据和知识用硬件进行快速的处理。

11.2.1.3 硬件后端机数据库机与知识库机

硬件后端机数据库机与知识库机与上面介绍的几种结构都不相同,它的模型如图 11.45 所示。

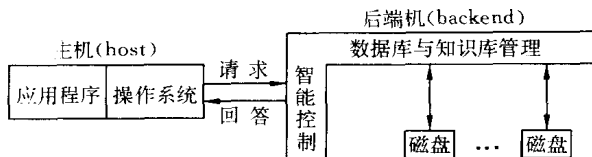


图 11.45 硬件后端机的数据库机与知识库机模型

在硬件数据库机与知识库机中,所有的数据和知识管理功能都在后端机上主要由硬

件实现。主机仅提供一个用户环境,包括提供操作系统、文件的编辑、各种软件的编译和输入输出工作等。因此,这种硬件后端机不是传统意义上的计算机,它需要采用特殊的系统结构,需要专门设计。

11.2.2 数据库机和知识库机在智能计算机系统中的作用

通常认为智能计算机(亦称为新一代计算机、第五代计算机等)主要由推理机、智能接口、数据库机和知识库机等组成。其中,可以把数据库机和知识库机想象为一个非常庞大的数据和知识的存储器及处理器,它能够采集、使用、存储和处理大量的数据和各种各样的人类知识。

目前,大部分数据库机和知识库机在整个智能计算机系统中仅作为后端机使用,它不直接面向用户。因此,一台数据库机或知识库机通常需要与一台前端机连接起来才能发挥作用,连接的方法可以有以下几种。

11.2.2.1 由数据库机和知识库机构成的智能计算机系统

数据库机或知识库机能够与推理机和智能接口一起构成一台智能计算机系统(intelligent computer system),如图 11.46 所示。

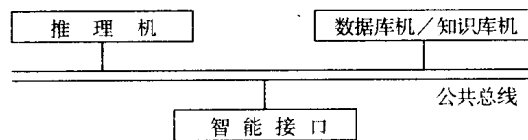


图 11.46 由知识库机等构成的智能计算机系统

数据库机或知识库机在整个智能计算机中的作用主要是存储大量的数据和知识,并对所存储的数据和知识进行快速的处理。

11.2.2.2 数据库机与知识库机作为网络系统中的结点机

数据库机与知识库机还可以连接到局域网上,使连接在网上的所有计算机用户都能够共享数据库机和知识库机,如图 11.47 所示。

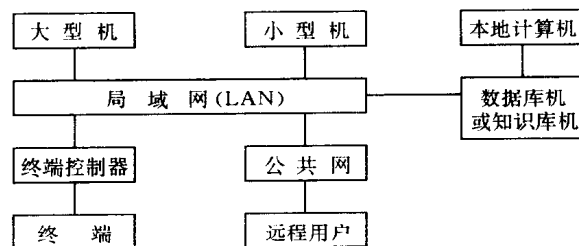


图 11.47 共享数据库机与知识库机的计算机系统

连接到网络上的可以有大型机、小型机、个人计算机、甚至是用户终端。局域网还可以连接到远程网上,使更多的用户共享数据库机和知识库机。连网之后,本地计算机(数据库

机或知识库机的主机)仍然可以使用原来属于它的数据库机和知识库机。

11.2.2.3 分布式数据库机与知识库机系统

用多台数据库机或知识库机的后端机可以连接成一个分布式数据库机系统 (DDBMS) 或分布式知识库机系统 (DKBMS), 如图 11.48 所示。

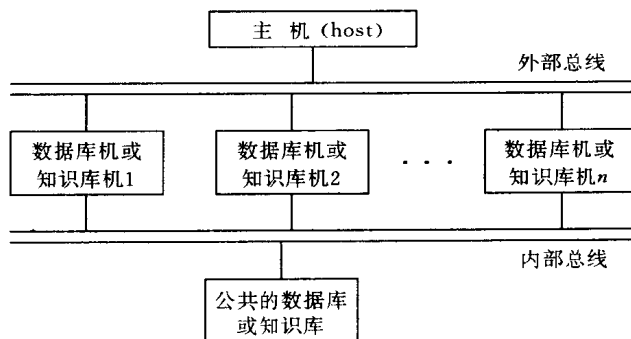


图 11.48 分布式数据库机系统与分布式知识库机系统

在整个分布式数据库机或知识库机系统中可以有一个集中的主机,也可以没有集中的主机,而完全采用分布方式管理。同样,可以有一个集中的数据库或知识库,也可以没有集中的数据库或知识库,完全使用分布于各个结点上的数据库和知识库。

11.2.3 数据库机与知识库机系统结构

数据库机与知识库机的结构设计应该主要考虑以下几点:

1. 具有快速查询数据和知识的能力。通常可以采用如下几种办法:

(1) 采用多个专用的处理部件并行工作。

(2) 在主存储器或磁盘存储系统的出口处设置一个专用的过滤器,用这个过滤器对原始的数据和知识进行预处理。

(3) 各部件之间采用紧密耦合方式充分开发细粒度并行性。

(4) 改进检索方法,提高检索速度。

2. 能够存储大量的数据和知识。通常采用的办法有:

(1) 内部存储器采用多级存储结构。这样做不仅能够增加存储容量,而且还能提高数据和知识的访问速度。

(2) 增加磁盘存储器的容量,或采用大容量的光盘存储器。

(3) 改进数据和知识的存储结构。

3. 采用模块化结构,以便能够充分利用 VLSI 技术。

4. 适当的软硬件功能分配。因为数据库机或知识库机最终必须由硬件和软件两部分共同组成,依靠硬件和软件的协同工作以达到性能和价格的平衡。

5. 提供良好的用户接口,使数据库机或知识库机能够与各种各样的前端机相连接,能让更多的用户访问它们。

11.2.3.1 数据库机与知识库机的逻辑结构

所谓逻辑结构(logical architecture)通常是指数据库机与知识库机在逻辑上要完成的功能,主要的功能有两大部分,一是转换部分(transaction),二是执行部分(execution),其逻辑结构可以用图 11.49 表示。

转换部分由三层组成,它们分别是通信层、内外格式转换层和操作形成层,这部分不涉及数据库和知识库。与数据库或知识库直接相关的只有操作执行层。

通信层负责与外部的通信联系,它接收和发送有关的信息,能够处理远程和本地的多种不同通信协议。

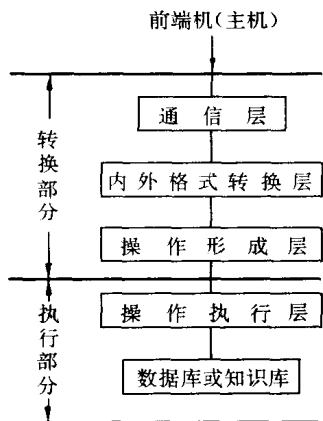


图 11.49 数据库机与知识库机的逻辑组成

内外格式转换层进行内部编码与外部格式之间的转换,它检验输入信息的合法性和操作码的正确性,它还提供通信层的调度,决定哪些信息需要转换,也决定哪些信息需要发送至下一层。

操作形成层接收来自内外格式转换层的信息,并将它们变换成一组数据库或知识库内部模式的操作码。

在转换部分的三层中,第一层处理外部信息而不必关心其它内容,第二层进行内部与外部格式的转换,第三层安排如何进行操作,只有这一层与数据库或知识库的内部模式有联系。

执行部分从逻辑上看很简单,它只需要执行上一层送来的操作命令即可。执行的结果返回给转换部分。

11.2.3.2 数据库机与知识库机的物理结构

与逻辑结构相对应,一种典型的数据库机或知识库机的物理结构(physical architecture)如图 11.50 所示。

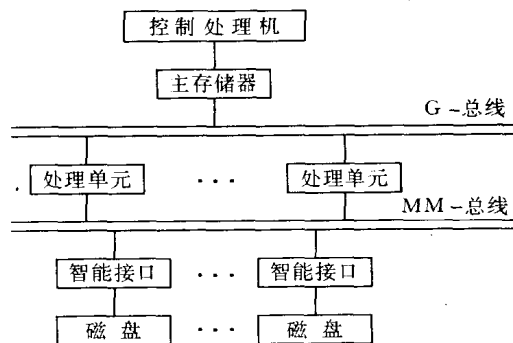


图 11.50 一种通用的数据库机与知识库机系统结构

在图 11.50 所示的系统结构中,处理单元、总线、存储器和磁盘系统都可以采用标准的硬件,以降低费用。当然,在需要高性能的场合,对于少量的特殊设备可以根据其功能要

求进行专门设计。下面,就其中的三个主要部件——处理机、存储器和智能接口的结构分别进行讨论。

11.2.3.3 多级存储器结构

在数据库机与知识库机中,内部存储器通常要采用多级存储结构。这样做不仅能加大存储容量,更主要的是能够提高数据和知识的处理速度。图 11.51 是一个典型的三级存储器结构框图。

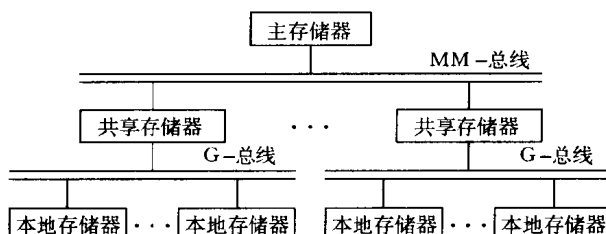


图 11.51 数据库机与知识库机中的一个三级存储器结构

图 11.51 中的本地存储器仅仅作作为某一个处理单元局部存储器,共享存储器为有关的处理单元和控制处理机所访问,只有主存储器能够被所有处理部件访问。

需要注意,增加存储器个数之后可能要增加访问存储器的冲突,当发生访问存储器冲突时系统的并行性必然要降低。因此,为了解决访问存储器的瓶颈问题,必须采用一种好的存储分配方案。

11.2.3.4 操作执行部件的结构

为了加快执行部件的工作速度,提高效率,在物理的实现上可以采用多个执行部件并行工作。图 11.52 就是与图 11.49 的逻辑结构中的执行部分相对应的物理结构。其中的执行部件可以采用传统的处理机,也可以采用专门设计的处理机。

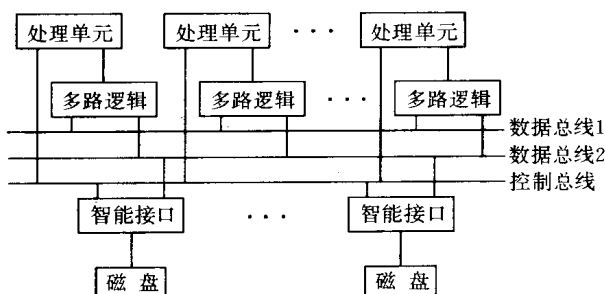


图 11.52 数据库机与知识库机中的执行部件

图 11.52 中的智能接口是一个特殊的部件,处理单元与磁盘设备之间的通信通过智能接口来实现。智能接口对从磁盘上读出的数据和知识进行预处理,这样可以加快知识的处理速度。

11.2.3.5 智能接口的结构

为了加快数据库机与知识库机的处理速度,在体系结构中可以加入一个或几个智能接口。当原始数据或原始知识从内存储器或磁盘设备上读出来时,这些智能接口可以完成对它们的关系运算。因此,智能接口的作用相当于一个预处理器,对原始数据和原始知识

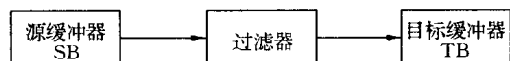


图 11.53 智能接口的结构

进行筛选,把符合条件的数据和知识送到下一个功能部件。智能接口的结构可以用图 11.53 来表示。

大家知道,对磁盘的读写操作相对于处理机的速度来说是很慢的,因此有必要在智能接口的输入端设置一个源缓冲器(SB),过滤器(filter)直接访问这个源缓冲器,这样可以加快读写的速度。过滤器产生的结果送至目标缓冲器(TB),处理单元只需要访问目标缓冲器即可。

结合图 11.50 看,SB 中的数据来源于磁盘存储器,TB 中的数据送至 MM-总线,经过处理单元进一步处理后再通过 G-总线送至主存储器,最后由请求该数据库机或知识库机的前端机取走,返回给用户。

在整个处理过程中,过滤器是一个非常重要的设备,是需要专门设计的一个硬件。过滤器的功能主要是处理一元和二元的关系运算,为了完成这些运算,在过滤器中必须装入相应的有限状态自动机(FSA;finite-state automaton)的描述,FSA 的产生可以用图 11.54 来表示。



图 11.54 过滤器的装入过程

一旦把 FSA 装入到了过滤器之后,过滤器就能够扫描 SB 中的关系了。如果是二元操作,SB 从逻辑上可以分为两个源缓冲器,即 SB1 和 SB2,每一个源缓冲器都装载一个关系。

如果 TB 已经满了而又不能立即将其中的数据送出去的话,必须把 TB 中的一部分重新写入到磁盘中。

从上面的介绍中可以看出过滤器是一个真正有智能的操作部件,它能够控制对磁盘的读写,能够初步筛选出符合条件的数据和知识。

过滤器的工作过程是这样的,如果前端机送给数据库机或知识库机的请求中包含有对数据或知识的条件操作,则需要对数据或知识进行预处理。于是,处理单元通过 MM-总线的控制线向智能接口发出命令。智能接口中的 CPU 接收到命令之后通过查询得到相应的关系格式,并根据关系格式和操作类型将其编译成有限状态自动机存放在过滤器的 CPU 中,同时发出控制信息到磁盘,把从磁盘中读出的数据送入 SB 中。过滤器对存放在 SB 中的数据进行扫描,一旦选出符合有限状态自动机条件的数据就送至 TB 中,以准

备由处理单元取走作进一步的处理。一旦 TB 满了,它就会向 CPU 发出请求,CPU 将其中的一块重新写入到磁盘中。

图 11.55 是法国 IMAG 实验室的 OPALE 知识库机中采用的智能接口。在 OPALE 知识库机中,连接到公共通信网上的每个主存储器(primary memory)和每个磁盘系统都要经过一个或几个这样的智能接口。

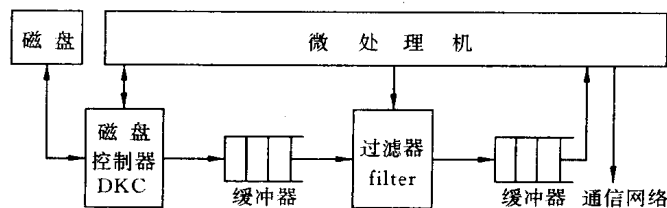


图 11.55 在 OPALE 知识库机中采用的智能接口

图 11.56 是 OPALE 知识库机中的关键部件过滤器的结构。这种过滤器主要由四个部件组成,其中包括传输控制器(TC:transfer controller)、词法自动机(LA:lexical automaton)、句法自动机(SA:syntax automaton)和结果选择器(RS:result selection)等。

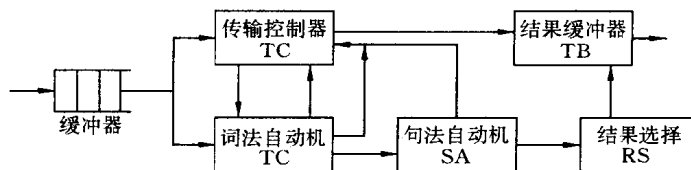


图 11.56 在 OPALE 知识库机的智能接口中采用的过滤器

11.2.4 后端机与前端机的接口

目前,大部分数据库机与知识库机并不能构成一个独立的计算机系统,它们通常要通过一个接口部件与前端机连接起来。数据库机或知识库机与前端机的连接关系如图11.57所示。

从图 11.57 中可以看出,数据库机或知识库机不仅能够被本地计算机访问,而且能够作为网络中的结点机。

在数据库机与知识库机的组成中,接口是很重要的一个部件,因为主机和数据库机或知识库机之间的信息传输都要通过它来完成。接口设计得好坏直接关系到数据库机与知识库机的工作效率。一个设计得好的接口可以节省大量的信息传输时间,能够为用户提供快速的响应。相反,一个设计得不好的接口将会造成大量信息的延迟,降低数据库机与知识库机的工作效率,使连接在系统中的数据库

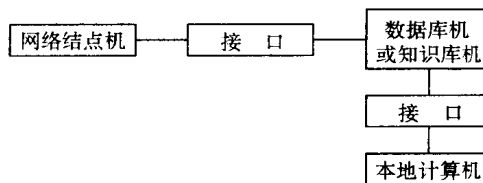


图 11.57 数据库机或知识库机通过智能接口与前端机连接

机和知识库机不能充分发挥其效能。

接口设计可以从逻辑和物理两个方面去探讨。在设计过程中主要关心的是接口的工作速度问题。在系统中有了快速的传输才可能有高的工作效率。

11.2.4.1 逻辑接口

从用户和程序员的角度看,前端机与数据库机或知识库机之间的接口是一种逻辑接口(logical interface)。从逻辑上可以把接口看作是一个断点(break point),通过这个断点前端机可以调用数据库机或知识库机的有关功能。对于这种断点可以采用下面四种方法来实现:

- (1) 端点用户查询语言(EUL:end-user query language)
- (2) 知识操纵语言(KML:knowledge manipulation language)
- (3) 逻辑输入输出语言(LIO:logical I/O language)
- (4) 物理输入输出语言(PIO:physical I/O language)

对于不同的实现方法,数据库机或知识库机被访问的次数是不同的。为了直观地讨论这个问题,下面举一个简单的例子。

例 11.5 查询在 A 系且年龄为 20 岁的学生姓名。

假设有一个简单的关于学生情况的数据库,其域的组成分别为学号、姓名、年龄、性别、所属系等,每个域的字节数是固定。

RELATION——student

FIELD——no #, name, age, sex, dept, ...

共有 50 个记录,它们按照系名域顺序地存放在数据库中的 100 到 104 页(page)中,每页 10 行(line)。

在这个例子中,采用不同层次的接口造成前端机与数据库机通信次数是不一样的。在 EUL 层只有一次,在 DML 层有 101 次,在 LIO 层有 51 次,在 PIO 层是 5 次。

在不同层次前端机完成的工作量也是不一样的,在 EUL 层前端机完成的工作量最少,而在 PIO 层前端机的工作量最大。

假设前端机与数据库机之间每次通信处理所用的时间为 1 个单位,信息处理的时间也为 1 个单位。从上面的叙述中可以看到,在 EUL 层通信处理和信息处理总共需要的时间为 4 个单位。在 KML 层,由于其通信处理的次数为 101 次,因此通信处理和信息处理的时间为 404(即 $101 \times 2 \times 2$)个单位时间。在 LIO 层,需要的通信次数为 51 次,因此通信处理和信息处理的时间总共为 204(即 $51 \times 2 \times 2$)个单位时间。在 PIO 层,需要通信的次数为 5 次,所以在通信处理和信息处理上所花费的时间为 20(即 $5 \times 2 \times 2$)个单位时间。可见在不同层次的接口执行时间有很大的差别,这是在设计前端机与数据库机或知识库机的逻辑接口时需要考虑的一个主要因素。

11.2.4.2 物理接口

从系统结构和系统实现的观点看,前端机与数据库机或知识库机之间的物理接口(physical interface)可分为三类,即通信接口(communication interface)、通道接口(chan-

nel interface)和共享存储器接口(shared memory interface)。

一般希望前端机与数据库机或知识库机之间的通信接口是通用的,这样做的主要好处是不同的前端机系统都能够与数据库机或知识库机相连接,能够直接访问其权限之内的数据和知识。

目前,许多通信接口都已经是标准化的了,并且在计算机网络中得到了广泛的应用。而通道接口在不同的计算机系统中是不一样的,如果将其应用于数据库机或知识库机中会造成后端机只能与某些特殊的计算机系统相连接。共享存储器接口的性能完全取决于前端机的结构,前端机结构的不同将造成数据库机或知识库机与存储器的接口也不相同。

在上面三种物理接口中速度最快的是共享存储器接口,因为它直接与前端机相连接。其次是通道接口,速度最慢的是通信接口。而标准化程度正好相反,通信接口的标准化程度最好、最通用。

11.2.4.3 后端机与前端机之间的接口选择

上面分别介绍了前端机与数据库机或知识库机的逻辑接口和物理接口,然而,逻辑接口与物理接口并不是截然分开的。物理接口的选择决定了逻辑接口的选择,而逻辑接口的选择又依赖于物理接口的选择,因此,必须从整个系统的角度来考虑接口的选择问题。总的来说,接口的选择应该遵循如下原则:

(1) 如果数据库机或知识库机是前端机的一部分,或者说数据库机或知识库机是前端机系统中的一个组成单元,那么物理接口应该采用共享存储器接口,逻辑接口确定在PIO层、LIO层或者DML层都是合适的。在这种情况下,数据库机或知识库机的接口取决于前端机系统。

(2) 如果数据库机或知识库机是几台相同结构计算机系统的共享资源,则物理上选择通道接口,而逻辑上采用PIO层或者LIO层都是合理的。

(3) 如果数据库机或知识库机是不同计算机系统的共享资源,则物理接口最好采用标准的通信接口,而逻辑接口最好采用EUL层。这样做虽然通信速度可能要降低些,但是后端机的应用范围却扩大了。

如前面所述,在大部分数据库机和知识库机系统中数据库机和知识库机是作为后端机的形式出现的,它们不是前端机系统的组成单元,也不是专门为某一种结构的前端机专门设计的。因此,对于这样的数据库机和知识库机,其逻辑接口采用EUL层是比较合理的,其原因如下:

(1) 在EUL层,接口采用了近似于自然语言的高层操纵语言,使各种各样的应用程序与数据库机和知识库机充分独立,从而更适合于用户的使用。

(2) 正如前面所分析的,在EUL层前端机与数据库机和知识库机之间的通信次数最少,所需要传输的信息量也最少,因此能提供更多的物理接口的选择。

(3) 在EUL层,前端机的任务只是组织一个命令格式,并将其传送给数据库机或知识库机,而数据库机和知识库机的任务是把处理的结果发送出来。因此,它们之间通信所需要的软件最少,从而小型机、个人计算机、甚至于一个智能终端都能够成为访问数据库机和知识库机的前端机。

对于物理接口,由于希望数据库机和知识库机能够被连接到网络上,因此最好能采用标准的通信接口,这样能够增加数据库机和知识库机应用的广泛性和通用性,使数据库机和知识库机能够发挥其最大效能。

选择前端机与数据库机和知识库机之间的接口是十分重要的,接口的好坏往往直接影响到整个系统的效能,必须加以足够的重视。这里所说的物理接口和逻辑接口是基于数据库机和知识库机的应用环境提出的,它能够接收来自不同地域的计算机系统的访问,同时也能被各种各样的计算机——如小型机、超级小型机、个人计算机和智能终端等访问,并且所需要的通信软件也很少。所有这一切都说明选择逻辑接口在用户语言查询(EUL)层,而物理接口采用标准的通信接口是合理的,而且选用标准接口还具有降低费用、易于扩展、便于维修等优点。

总之,接口的选择一定要符合用户的要求,并且按照实际情况进行选择,只有这样才能提出既让用户满意又切合实际的逻辑接口和物理接口。

11.2.5 典型的数据库机与知识库机

数据库机的研究开始于七十年代初。1972年投入运行的XDMS实验数据库管理系统是第一个实现的软件后端机数据库机。同年,英国的ICL公司推出了CAFS数据库机,在这台数据库机中采用了具有相联检索功能的磁盘控制器。1980年IDM500智能数据库机研制成功。在此以后又出现了多种数据库机。

近几年来出现的知识库管理系统基本都是关系数据库系统和以关系代数为基础的推理机制的结合体。同样,知识库计算机的组成也往往是关系数据库计算机和关系代数推理机的结合。

知识库机的研究工作开始于八十年代。1984年日本ICOT研究所推出实验型知识库机PSI+Delta,它由个人串行推理机PSI和关系数据库机Delta通过局域网连接而成。以后,国内外提出了多种知识库机的设计方案。

在表11.1中给出近几年来国内外知识库机的研究情况。

表 11.1 典型知识库机的情况统计

系统名称	研制单位	知识表示方法	结构耦合程度
PSI+Delta	日本 ICOT 研究所	Prolog+关系代数	松散耦合
Relational KBM	日本 ICOT 研究所	关系知识库模型	松散耦合
Surrogate File	美国 Syracuse 大学	Prolog	松散耦合
GRM	英国 Strathelyde 大学	多继承性网络 MIN	紧密耦合
OPALE	法国 IMAG 实验室	Prolog	松散耦合
HAS	德国西门子子公司	Prolog	紧密耦合
PUKBM	清华大学计算机系	Prolog+unify	紧密耦合
RKBM	国防科技大学	Prolog	紧密耦合

11.2.5.1 高速数据处理技术

数据库机与知识库机的关键技术是对数据进行快速的处理和传送。为了提高数据库机和知识库机的性能,必须研究高速数据处理技术。目前,经常采用的高速数据处理技术有如下四种:

(1) 旋转处理(process on the fly)技术

在数据库机和知识库机中,大量的数据都存放在磁盘中。为了加快数据处理的速度,减少数据传送,在磁盘旋转过程中就对数据进行处理,这种方法叫做旋转处理技术。1970年 L. Slotnick 提出逻辑磁道(logic per-track)方案。在固定头磁盘的每条磁道上都配置一个微处理机,当磁盘旋转一周时就能扫描到全部数据,并在扫描过程中直接对数据进行处理。逻辑磁道技术能够有效改善信息检索和字符串处理的速度。后来,在 CASSM、RAP、CAFS 等数据库机中都采用了这种技术。

(2) 散列位阵列(hash bit array)技术

关系数据库机中的连接运算所需要的时间很长,所需时间与记录数目的平方成正比。通过散列位阵列技术可以实现半连接运算,能够比较好地解决这个问题。例如要对关系 R、S 执行半连接运算,首先将位阵列中的全部数位清“0”,然后对关系 R 中指定属性的全部值进行散列运算,并按照其结果给出的地址将相应位阵列单元置“1”。对全部 R 操作做完之后,接着采用同样的散列函数对关系 S 中指定属性的全部值进行散列运算,如果原来位阵列单元是“1”,则结果包含 S,否则不包含。

(3) 相联处理(associative process)技术

采用相联处理机和相联存储器能够有效提高数据的处理速度。相联处理主要适用于字符串及字符串之间的关系运算。按照内容来检索数据,其处理的效率高。相联处理技术已经应用于 CAFS、DIALOG 和 RDBM 等数据库机中。

(4) 多处理机技术

使用多个处理机能够在同一时刻或同一时间间隔内完成两种或两种以上性质相同或不同的运算,以提高数据处理的速度。根据处理机之间的不同连接方式,可以分为紧密耦合和松散耦合两种类型。紧密耦合采用总线、交叉开关、多端口存储器和开关枢纽等结构,如 DIRECT 数据库机采用交叉开关结构。松散耦合一般通过互连网络实现分布式连接,如 DBMAC 数据库机等。

11.2.5.2 数据库机的分类

根据数据库机中使用的处理机个数和是否在磁盘存储器中直接进行检索操作,可以把数据库机分为以下 5 种类型:

(1) 单处理机间接检索型

用一台通用的处理机作为主机的后端机,在后端机上利用固件实现数据库管理的功能。如智能数据库机 IDM500 属于这种类型。

(2) 单处理机直接检索型

采用专用处理机直接实现数据的检索功能。从磁盘存储器中读出的数据直接送到相

联检索部件和记录检索部件进行数据筛选,直接获得符合要求的结果。如 CAFA 数据库机属于这种类型,它还采用散列位阵列技术提高检索效率。

(3) 多处理机直接检索型

采用多处理机组成数据库机,并且直接在磁盘存储器上进行数据检索,这种数据库机的磁盘存储器一般采用固定头磁盘或磁盘阵列机等实现。在每条磁道上都安装有一个处理机,在磁盘转动过程中直接对数据进行处理,检索出符合要求的数据。这种类型的数据库机有 RAP 等。

(4) 多处理机间接检索型

先将数据从磁盘存储器读到缓冲存储器,然后采用多个处理机进行并行处理,实现多指令流多数据流操作。如 DIRECT 数据库机属于这种类型。

(5) 多处理机组合检索型

由多个功能专用的处理机组合起来实现数据库机,完成数据的并行检索。如俄亥俄州立大学提出的 DBC 数据库机属于这种类型。

11.2.5.3 Relational KBM 关系知识库机

关系知识库机 Relational KBM 是由日本新一代计算机技术研究所(ICOT; Institute for New Generation Computer Technology)研制的,它的体系结构如图 11.58 所示。

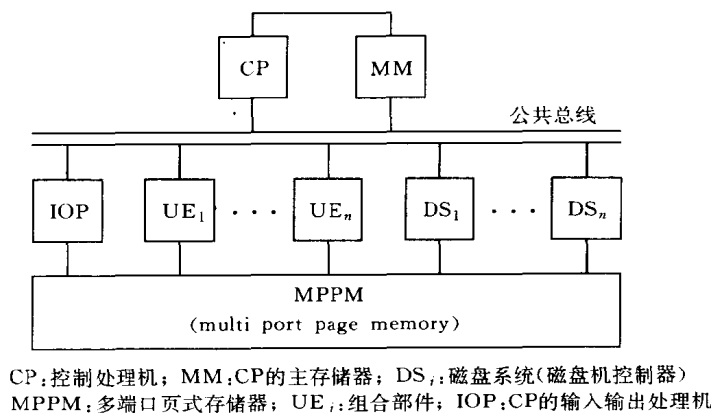


图 11.58 Relational KBM 知识库机结构

Relational KBM 知识库机主要由一组组合部件 UE、一组磁盘系统 DS、一个控制处理机 CP、一个主存储器 MM、一个输入输出处理机 IOP 和一个多端口页式存储器 MPPM 组成。每个 UE 和 DS 都直接与 MPPM 相连接。

Relational KBM 知识库机的主要优点是组合部件与磁盘系统之间有很宽的频带。磁盘系统产生数据流并把它发送到 MPPM,组合部件从 MPPM 得到数据流,采用流水线方式处理后再把结果数据流送回 MPPM。另外,保存在磁盘系统中的结果可以再次为组合部件使用,也可以输出给用户。

控制处理机不仅控制数据流,而且也控制各个部件的并行执行。它能够发送命令到与 MPPM 连接的组合部件和磁盘系统,并且从那里得到回答。然而,因为 MPPM 设计成能

够同时传送多页数据,所以它不适合快速传送少量数据。

命令的发送和接收都通过控制总线,主存储器、控制处理机所属的输入输出处理机和其它部件都连接到这条控制总线上。控制处理机通过输入输出处理机直接与 MPPM 连接,当控制处理机需要访问 MPPM 时,必须通过这个输入输出处理机进行,而在其它时间输入输出处理机用作与主机的接口。

组合部件是为项到项的关系检索而专门设计的一种硬件,每个组合部件有三条通路或多端口页式存储器相连接,其中两条通路用来从多端口页式存储器中读取数据,剩下的一条通路用来把结果写回到多端口页式存储器中。组合部件采用流水线方式处理数据流,它从输入通道得到数据流,并把处理结果返回到输出通道,它可以有条件地从数据流中检索出目标项来。

多端口页式存储器 MPPM 由一组输入输出端口、一组存储器堆和一个把这些输入输出端口和存储器堆连接起来的开关网络组成,如图 11.59 所示。

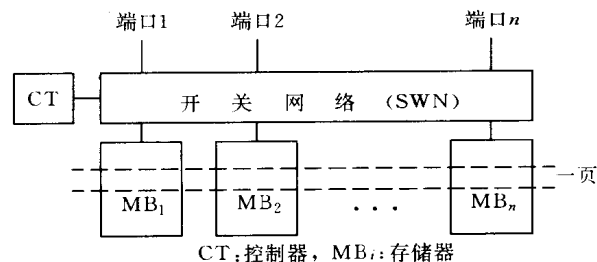


图 11.59 多端口页式存储器结构

11.2.5.4 PSI+Delta 知识库机结构

由日本新一代计算机技术研究所 ICOT 的 K. Murakami 等人研制的实验性知识库机如图 11.60 所示,它主要由个人串行推理机 PSI 和关系数据库机 Delta 通过局域网 LAN 连接而构成。

该知识库机的工作过程是这样的:如果用户在 PSI 机上编写一个 Prolog 程序查询关系数据库机 Delta,按照用户和 Prolog 程序所建立的查询内容,由 PSI 机的软件产生 Delta 机的查询计划;然后把查询计划转换成软件包形式的 Delta 机命令,这个命令通过 LIA 网络接口和 LAN 局域网将查询结果送到 Delta 机。Delta 机从软件包中抽取 Delta 机的命令,在分析了这些检索命令之后,Delta 机从关系数据库中检索出有关数据,通过 LAN 局域网把数据送到主机 PSI 上。

下面主要介绍 Delta 机的系统结构,如图 11.61 所示。

关系数据库机 Delta 由关系数据库管理和处理子系统 RSP 及多级存储器子系统构成,其中 RSP 子系统负责整个系统的控制、监督和操作处理,多级存储器子系统负责关系

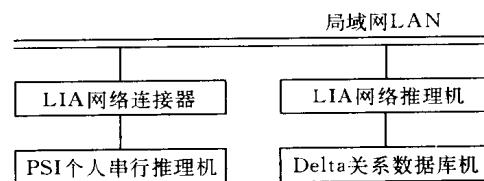
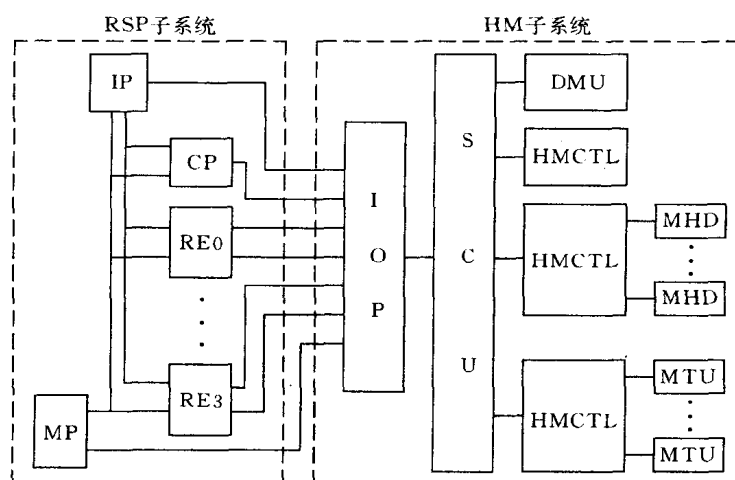


图 11.60 由 PSI 和 Delta 组成的知识库机系统结构



RSP:关系数据库管理和处理子系统, HM:多级存储子系统, IP:接口处理机
CP:控制处理机, MP:维护处理机, RE:关系数据库机, IOP:输入输出处理机
SCU:存储器控制单元, DMU:数据存储单元, HMCRL:多级存储子系统控制器
MHD:活动头磁盘, MTU:磁带控制器

图 11.61 关系数据库机 Delta 的结构

数据库的存储和检索等工作。

RSP 子系统由控制处理机 CP、接口处理机 IP、维护处理机 MP 和 4 个关系数据库机 RE 组成,每个部分都包括有软件和硬件。

每个部件基本上都由一个处理机和 512K 字节或 1M 字节的局部存储器构成。为了提高存储容量,在控制处理机 CP 内部提供 15M 字节的半导体存储器。每个 RSP 单元通过三条 IEEE488 总线连接到另一个 RSP 单元上。在每个 CP、IP 和 MP 内部都设置有与多级存储器子系统 HM 连接的适配器,在每个 RE 内部有两个这样的适配器,其中一个做输入,另一个做输出。

图 11.61 中的 4 个关系数据库处理机 RE0 至 RE3 是关系数据库计算机 Delta 的核心部件,它们需要用专门用硬件实现。一个 RE 的结构如图 11.62 所示。

HM 子系统有一个非易失性的高速数据存储单元 DMU,存储容量为 128M 字节,磁盘存储器的最大容量为 20G 字节。

11.2.5.5 PUKBM 知识库机

PUKBM 知识库机是用一台多处理机系统改装而成的,它采用 PROLOG 逻辑程序设计语言和 UNIFY 关系数据库管理系统两者结合的模式管理知识库,因此取名叫 PUKBM。其中,用 PROLOG 管理规则库,用 UNIFY 管理事实库,在整体上用关系代数作为推理机制,它的组成如图 11.63 所示。

PUKBM 知识库机共由 8 个处理机组成,其硬件结构采用功能分布的紧密耦合方式,各个处理机在功能上是独立的,有一个处理机作主机,有两个处理机作后端机,它们分别处理事实和规则,在这两个后端机与主机之间有一个处理机专门负责通信等工作。另外,

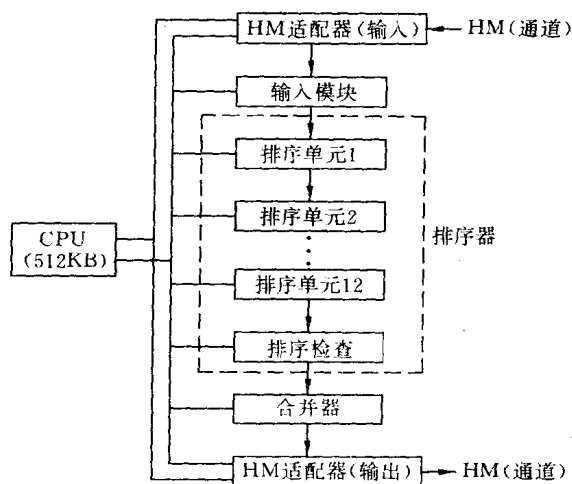


图 11.62 关系数据库处理机 RE 的结构

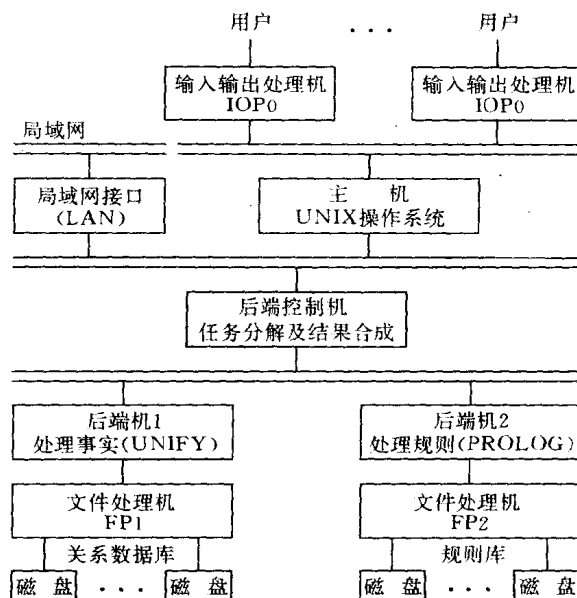


图 11.63 PUKBM 知识库机的系统结构

有两个输入输出处理机负责与用户的连接及其它输入输出工作,有两个处理机作文件处理机,对从磁盘上读出的数据和规则进行预处理。

这种多处理机结构的知识库机有两个明显的优点,其一是硬件上比较容易实现,因为每个处理机都是功能单一的专用处理机;其二是软件上有比较成熟的技术可以借鉴,例如,运行在主机上的 UNIX 操作系统,运行在两个后端机上的 PROLOG 程序设计语言和 UNIFY 关系数据库管理系统等。

在整个知识库机系统中只有主机是面向用户的,它安装有通用的 UNIX 操作系统,

可以运行各种各样的用户程序。主机通过两个输入输出处理机 IOP1 和 IOP2 连接大量用户终端和打印机等。另外,还可以通过局域网接口连接更多的用户,使网络上的其它计算机用户都能够共享这个知识库机。

后端控制机位于主机、局域网接口和两个后端机中间,它的主要工作有两项,一是负责通信工作,二是完成任务的分解与结果的合成。后端控制机接收从主机或局域网上送来的任务,把任务分解为事实和规则两部分,并把这两部分分别交给下一层的两个后端机进行处理。两个后端机的处理结果也要在后端控制机上合成,并转换成可以输出的形式分别送往本地的主机或局域网上的其它计算机。另外,运行在后端控制机上的所有软件都要专门编写,没有现成的软件可以借鉴。

后端机 1 专门处理事实。在后端机 1 上安装 UNIFY 关系数据库管理系统,另外也安装有专门编写的通信软件和数据处理软件,它通过文件处理机 FP1 连接 4 台磁盘作为关系数据库。

后端机 2 专门处理各种各样的规则,安装 PROLOG 程序设计语言,另外也安装有通信软件和规则处理软件,它通过文件处理机 FP2 连接 4 台磁盘作为规则库。在后端机 2 中处理的规则采用关系代数形式。

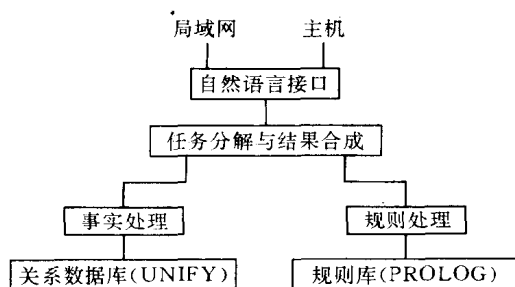


图 11.64 PUKBM 知识库机的逻辑结构

PUKBM 知识库机的逻辑结构经过简化后如图 11.64 所示。其中,最下面的两层直接用 PROLOG 程序设计语言和 UNIFY 关系数据库实现。为了扩大知识库机的应用范围,在原有 PROLOG 语言的基础上又增加一些谓词。上面三层,即规则处理、事实处理、任务分解及结果合成、用户接口这四部分软件都是专门编写的。

因为处理规则和处理事实的两个后端机都不直接面向用户,它必须通过后端控制机的接口与主机或局域网相连接,因此后端机的接口是知识库机与外界联系的唯一窗口,这个窗口也称为用户接口。用户接口的设计目标应当尽量满足各种不同用户的需要,给用户提供一个尽可能友好的界面。

目前,自然语言接口是比较理想的用户接口,它有两个最主要的优点。一是自然语言接口只要求用户描述要解决的问题,而不需要用户描述如何解决问题,这一点与目前普遍采用的大多数高级语言及用户接口均不相同;二是自然语言接口通常采用人们习惯的自然语言来描述问题,而不像大多数高级语言那样用数学语言来描述问题。但是,直到目前为止还没有一种比较好的知识操纵语言,因此,也更谈不上有一种好的知识库自然语言接口。在 PUKBM 知识库中所采用的自然语言接口还比较简单,还只是自然语言接口的一个子集,或称为受限自然语言接口。它的主要功能有:知识采集、知识查询、知识删除、知识更新和知识修改等。

任务分解与结果合成这一层接收自然语言接口送来的任务,把任务分解为数据服务子任务和规则服务子任务两个部分,并通过处理机之间的通信把这两个子任务分别交给

关系数据库后端机和规则库后端机去执行。两个后端机执行完成之后把结果返回来,在这里两种结果经过合成处理后以自然语言的形式发送给前端机。另外,在这一层中还要完成数据库和规则库的保护,如维护数据和规则的一致性、防止不合权限的及非法的访问、防止规则的重复定义等。

同样,通信工作也很重要。在各个层次之间都要进行通信,而且,根据层次的高、低采用不同的通信协议。

数据处理和规则处理是整个知识库机的关键部分,这两部分充分利用 PROLOG 程序设计语言和 UNIFY 关系数据库管理系统来完成,并利用这两个软件所提供的接口函数完成数据和规则的各种操作功能。

11.3 面向函数程序设计语言的归约机

11.3.1 引言

函数程序设计语言(functional programming language),简称函数式语言,是一种具有优美数学性质的说明性语言(declarative language)。它的根源可以追溯到 20 世纪 30 年代由 A. Church 发展起来的 λ 演算(λ -calculus)理论。 λ -calculus 是一种研究可计算性的基础理论,它尤其注重定义精确的符号系统来描述可以用机械方法或算法进行求值的操作功能和数据。Church 从这项研究得到的主要结论是:任何复杂的可计算的操作功能都可以用一些简单的计算操作表示出来,并且给出了通用的、机械性的规则用以处理这些简单的计算操作和由它们组成的表达式。在 λ -calculus 中,简单操作组成的表达式所代表的可计算性操作等价于数学意义上的函数。所以说, λ -calculus 的重要性体现在它定义了一种有效的描述函数的方法,并给出了对于这种描述方法的机械求值规则和有关定理。基于这样的思想,人们考虑发展更面向人的函数描述方法,用于表示各种可计算性问题,用计算机来完成计算处理过程。这正是函数式语言发展的源泉和理论基础。

从上可以看出,函数式语言程序的主要功能在于定义函数操作,设计函数语言就是确定如何用一套简洁有效的方法来说明各种可计算问题的操作功能。与传统的 Von Neumann 风格的程序设计语言相比,函数式语言是面向问题的(problem-oriented),它将描述问题是“干什么”的,它是一种说明性语言。而传统的程序设计语言是面向计算机的状态变化的,它描述计算机是“怎么干”来处理问题的,它是一种命令式语言(imperative language)。

函数式语言具有很多优点,主要表现为三个方面:

(a) 它具有更强的程序设计能力。由于它是面向问题的说明性语言,使得程序设计非常灵活和方便,程序结构简洁,错误减少,保证了大型复杂问题的程序设计的可靠性。

(b) 由于函数定义具有优美的数学性质,它非常适于问题的形式描述(specification)和程序的形式变换(transformation),是软件工程较为理想的工具。

(c) 在函数式语言中没有描述计算操作的顺序和机器状态,消除了副作用(side-effect),使之易于开发和管理程序中的并行计算潜力,提高程序的执行效率。

因此,函数式语言可以在许多领域,特别是人工智能领域得到广泛的应用,成为程序

设计语言研究中的一个有生命力的发展方向。

函数式语言的先驱是 60 年代早期 J. McCarthy 提出的 LISP 语言,30 余年以来,它已发展为一个拥有诸如 LISP、FP、ML、SASL、Lucid、HOPE、KPC 和 Miranda 等多种程序设计风格语言的庞大家族。

在早期的函数式语言研究中,研究工作的重点集中在语言的语法、语义和应用上。随着函数语言的广泛应用,随着并行计算机研究的深入,人们发现传统的 Von Neumann 计算机存在许多根本性的问题,函数式语言难于在传统计算机上高效实现。特别是不能开发计算的并行潜力,极大地限制了函数式语言在大系统问题中的应用。因此,函数式语言实现方法和直接支持函数式语言的计算机的研究得到越来越多的重视,它将在计算模型和系统结构上彻底摆脱传统的 Von Neumann 计算模型和系统结构概念和技术,从而成为新一代计算机研究领域的重要组成部分。世界各国的研究者们对函数式语言的有代表性研究有美国的 DAPS 系统、Rediflow 系统和 TTDA 系统,英国的 ALICE 系统、GRIP 系统和 Flagship,日本的 PIM-R 系统和 PIM-D 系统,荷兰的 DPRM 系统等等。

下面首先讨论最基本的归约计算方法,通过对从 λ -calculus 中基本执行规则发展起来的各种归约方法进行比较和分析,对为什么要采取组合子图归约进行讨论。接着讨论组合子图归约的实现技术,特别是并行执行组合子图归约的控制和管理机制,设计并行图归约计算模型的一些主要问题。最后,对基于并行图归约计算机系统结构设计问题进行分析 and 讨论。

11.3.2 函数式语言的归约计算方法

归约计算“reduction evaluation”是 λ -calculus 中最一般的、也是唯一的函数求值方法。从数学意义上看,一个复杂的函数表达式等价于其结果值,所谓归约就是不断化简表达式直至最简表达式的过程,这时得到的不能继续化简的表达式就是最初表达式的计算结果。在这一部分中,将以 λ -calculus 中的基本项(term)为对象,讨论归约计算方法及其实现效率。这是研究函数语言并行图归约模型的理论基础和出发点。

1. 归约基语言(reduction base language)

归约基语言是描述归约计算对象和执行过程的工具,归约计算可以看成是它语义的解释。一般的函数式高级语言也可以描述计算的对象和执行过程,但它们在描述形式和结构上千差万别。为了用统一的方式描述各种函数式语言中函数性成分的归约计算,为了更深入地研究归约计算的性质,一般采用最基本的 λ 表达式(λ -expression),在 λ -calculus 中又称为项,来作为研究的对象。任何可用函数式高级语言描述的计算对象都可以用 λ 表达式(简记为 λ -exp)来表示,所以称之为基语言。

λ -exp 的语法定义如下:

```
<λ-exp> ::= <constant>
| <variable>
| <λ-exp1> <λ-exp2>
| (λ<variable>. <(λ-exp)>)
```

其中, $\langle \text{constant} \rangle$ 包括任何代表确定含义的标识符, 如常数、原始函数操作“IF”、“+”、“-”等等。 $\langle \text{variable} \rangle$ 代表具有不确定含义的标识符, 在归约计算过程中它可以, 也仅可以一次被任何别的 $\lambda\text{-exp}$ 替代。 $\langle \lambda\text{-exp}_1 \rangle \langle \lambda\text{-exp}_2 \rangle$ 表示了 $\lambda\text{-exp}$ 之间一种特定的关系, 称为作用(application)。这是 $\lambda\text{-exp}$ 中唯一的操作关系, 它可以理解为函数 $\langle \lambda\text{-exp}_1 \rangle$ 作用到参量 $\langle \lambda\text{-exp}_2 \rangle$ 上的结果, 即 $\langle \lambda\text{-exp}_1 \rangle \text{ apply } \langle \lambda\text{-exp}_2 \rangle$ 。 $(\lambda \langle \text{variable} \rangle. \langle \lambda\text{-exp} \rangle)$ 称为 λ 抽象($\lambda\text{-abstraction}$), 它表示了一个函数的定义(匿名函数), 其中 $\langle \text{variable} \rangle$ 是形参, $\langle \lambda\text{-exp} \rangle$ 是函数体。在 $\lambda\text{-abstraction}$ 中, 函数体中出现的形参称为受囿变量(bound variable), 其余的变量都称为自由变量(free variable)。

在 $\lambda\text{-exp}$ 中, 所有的函数操作(包括原始函数操作)都采用前缀表示形式, 这是为了满足 $\lambda\text{-exp}$ 中作用关系的语义。作用关系要求操作必须放在参量的前面。例如, 如下的表达式都是合法的 $\lambda\text{-exp}$: 1 , $(+1)$, $(+1\ 2)$, $(\lambda x. (+xy))$, $(\lambda x. (\lambda y. +xy))1\ 2, \dots$ 等。

函数式语言的基本描述成分、函数、函数的参量和函数作用到参量上的结果等, 都可以统一地用 $\lambda\text{-exp}$ 来表示。因此, 以 $\lambda\text{-exp}$ 为归约计算对象, 研究 $\lambda\text{-exp}$ 的计算方法, 将揭示一般函数性语言核心执行机制的本质。在后续各节的讨论中, 所有的计算对象均指 $\lambda\text{-exp}$ (有时简称表达式), 并以 $\lambda\text{-calculus}$ 中对 $\lambda\text{-exp}$ 的计算方法为出发点, 分析和讨论归约计算方法的发展历程。

2. $\lambda\text{-exp}$ 的基本计算方法

在 $\lambda\text{-calculus}$ 中, 无论是待计算的对象还是计算的结果都表示为 $\lambda\text{-exp}$, 所以 $\lambda\text{-exp}$ 的求值操作就是 $\lambda\text{-exp}$ 的转换过程。通过转换, 一个 $\lambda\text{-exp}$ 变成一个语义上相等的新的 $\lambda\text{-exp}$, 不断重复这个过程, 直到 $\lambda\text{-exp}$ 不能继续转换为止, 这时的 $\lambda\text{-exp}$ 称为正规式(normal form), 它就是计算的结果。在 $\lambda\text{-calculus}$ 中, 用三条转换规则(conversion rules)来描述 $\lambda\text{-exp}$ 的机械转换过程, 它们分别称为 α 规则, β 规则和 η 规则。人们常用式: $\langle \lambda\text{-exp}_1 \rangle \text{ CON}_i \langle \lambda\text{-exp}_2 \rangle$ 表示通过 i (α 或 β 或 η) 规则可以把 $\lambda\text{-exp}_1$ 转换为 $\lambda\text{-exp}_2$, 据此可以定义三条转换规则如下:

α : 如果在 $\langle \lambda\text{-exp} \rangle$ 中没有变量 y 出现, 则

$$(\lambda x. \langle \lambda\text{-exp} \rangle) \text{ CON}_\alpha (\lambda y. \langle \lambda\text{-exp} \rangle [y/x])$$

β : $(\lambda x. \langle \lambda\text{-exp} \rangle) \langle \lambda\text{-exp} \lambda \rangle \text{ CON}_\beta \langle \lambda\text{-exp} \rangle [\langle \lambda\text{-exp}' \rangle / x]$

η : 如果在 $\langle \lambda\text{-exp} \rangle$ 中没有变量 x 出现, 则

$$(\lambda x. \langle \lambda\text{-exp} \rangle x) \text{ CON}_\eta \langle \lambda\text{-exp} \rangle$$

其中 $\langle \lambda\text{-exp} \rangle [y/x]$ 或 $\langle \lambda\text{-exp}' \rangle [\langle \lambda\text{-exp}' \rangle / x]$ 表示将 $\langle \lambda\text{-exp} \rangle$ 中所有的变量 x 都替代为 y 或 $\langle \lambda\text{-exp}' \rangle$ 。

从上可知, α 规则实质上是换名变换, 目的是为了表达式中的变量名冲突(name crash)。 η 规则是 β 规则的一种特殊情况, 因为, 如果在 $\langle \lambda\text{-exp} \rangle$ 中没有变量 x 出现, 则有:

$$\begin{aligned} (\lambda x. \langle \lambda\text{-exp} \rangle x) \langle \lambda\text{-exp}' \rangle & \text{ CON}_\beta (\langle \lambda\text{-exp} \rangle x) [\langle \lambda\text{-exp}' \rangle / x] \\ & = \langle \lambda\text{-exp} \rangle \langle \lambda\text{-exp}' \rangle \end{aligned}$$

所以说, η 规则是一种优化变换。

β 规则是最本质的 $\langle \lambda\text{-exp} \rangle$ 转换规则, 用 β 规则转换 $\lambda\text{-exp}$ 的过程就称为归约。通过

上面 β 规则的定义可以看到 λ -exp 归约计算的两个重要特性:

(a) 只有对特定的 λ -exp 才能进行归约。

定义 1 若 λ -exp 满足:它是一个作用,形为 $\langle \lambda\text{-exp}_1 \rangle \langle \lambda\text{-exp}_2 \rangle$, 并且 $\langle \lambda\text{-exp}_1 \rangle$ 是一个 λ -abstraction, 则称这个 λ -exp 为一个可归约表达式 redex (reducible expression)。

根据 β 规则的要求, 只有 redex 才能进行归约变换。显然, 不含有任何 redex 的 λ -exp 就是正规式。

(b) 归约操作需要对受囿变量进行替代—— β 替代 (β -substitution)——才能产生新的 λ -exp。形如 $\langle \lambda\text{-exp} \rangle [\langle \lambda\text{-exp}' \rangle / x]$ 这样的操作就是 β 替代。

例 11.6 对于表达式 $(\lambda x. (\lambda y. + (* x x) (* y y))) 2 3$, 其归约过程如下:

$$\begin{aligned} & (\lambda x. (\lambda y. + (* x x) (* y y))) 2 3 \\ & \text{CON}_\beta (\lambda y. + (* x x) (* y y)) [2 / x] 3 \\ & = (\lambda y. + (* 2 2) (* y y)) 3 \\ & \text{CON}_\beta (+ (* 2 2) (* y y)) [3 / y] \\ & = (+ (* 2 2) (* 3 3)) = 13 \end{aligned}$$

β 规则确定了 λ -exp 归约计算的基本方法, 但如何机械地使用这个规则来归约各种 λ -exp 呢? 这个问题包括两个方面:

(a) 在一个表达式中可能出现多个 redex, 选择不同的 redex 归约将得到不同的新表达式。redex 的选择策略将直接影响归约的执行进程, 它反映了表达式的语义。所以, 这是语义研究的问题 (semantic issues)。

(b) 对于一个选定的 redex, 可以用多种方法进行变量替代 (β 替代)。不同的替代方法虽不影响表达式的语义, 但将极大地影响归约过程的执行效率。所以, 这是语用研究的问题 (pragmatic issues)。

下面分别对这两方面的问题进行讨论。

3. 归约计算的语义研究

在 λ -calculus 中, 选择 redex 的策略——亦称为计算序 (evaluation order)——有两种: 正规序 (normal order) 和作用序 (applicative order)。

根据定义 1 可知, 任一 redex 的首 (head) 必是“ λ ”。如果 redex₁ 的首“ λ ”处在 redex₂ 的首“ λ ”的左部 (不一定相邻), 则称 redex₁ 比 redex₂ 更左。在一个 λ -exp 中, 如果选取最左的 redex 进行归约, 则称为正规序归约; 如果选取最右的 redex 进行归约, 则称为作用序归约。

例 11.7 对表达式 $(\lambda a. (\lambda y. (\lambda z. y z)) a) (\lambda x. x) p$, (λ -exp 采用左结合优先方式), 分别用两种序归约, 过程如下:

正规序:

$$\begin{aligned} & (\lambda a. (\lambda y. (\lambda z. y z)) a) (\lambda x. x) p \\ & \underbrace{\hspace{1.5cm}}_{\text{redex}} \\ & \Rightarrow (\lambda y. (\lambda z. y z)) (\lambda x. x) p \\ & \underbrace{\hspace{1.5cm}}_{\text{redex}} \end{aligned}$$

$$\begin{aligned}
 &\Rightarrow (\lambda x. (\lambda x. x)z)p \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow (\lambda x. x)p \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow p
 \end{aligned}$$

作用序：

$$\begin{aligned}
 &(\lambda a. (\lambda y. (\lambda z. yz))a)(\lambda x. x)p \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow (\lambda a. (\lambda z. yz))(\lambda x. x)p \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow (\lambda a. (\lambda x. x)z)p \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow (\lambda z. z)p \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow p
 \end{aligned}$$

(其中“ \Rightarrow ”表示使用 β 规则进行归约变换)。

关于计算序的研究,Church-Roose 定理已经证明了如下的重要结论：

- 对于同一表达式,如果用不同的序进行归约都能得到正规式,那么它们必是相同的正规式；
- 对于一个表达式,如果它能归约为正规式,那么一定可以用正规序归约得到这个正规式。

前一条结论使得在归约计算中可以运用不同的序进行归约,特别是对多个 redex 同时进行归约,保证得到唯一的、正确的结果。这是并行归约计算方法的理论基础。后一条结论指出,正规序归约是安全的,它将保证能得到归约的结果,即归约一定是可停机的。而作用序归约有可能是不终止的,如下例所示。

例 11.8 对于表达式 $(\lambda x. a) ((\lambda y. y y) (\lambda z. z z))$ 分别用二种序进行归约：

正规序：

$$\begin{aligned}
 &(\lambda x. a) ((\lambda y. y y) (\lambda z. z z)) \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\Rightarrow a[(\lambda y. y y)][(\lambda z. z z)/x] \\
 &= a
 \end{aligned}$$

归约是停机的,得正规式 a 。

作用序：

$$\begin{aligned}
 &(\lambda x. a) ((\lambda y. y y) (\lambda z. z z)) \\
 &\quad \underbrace{\quad \text{redex} \quad} \\
 &\quad \underbrace{\quad \text{redex} \quad}
 \end{aligned}$$

$$\begin{aligned}
&\Rightarrow (\lambda x. a)(y y)[(\lambda z. z z)/y] \\
&= (\lambda x. a) \underbrace{((\lambda z. z z)(\lambda z. z z))}_{\text{redex}} \\
&\Rightarrow (\lambda x. a)(z z)[(\lambda z. z z)/z] \\
&= (\lambda x. a) \underbrace{((\lambda z. z z)(\lambda z. z z))}_{\text{redex}} \\
&\Rightarrow \dots\dots
\end{aligned}$$

显然,归约将不会终止。

作用序和正规序归约的根本区别是函数作用时对参量的处理方法不同。作用序归约要求先归约参量,然后归约函数作用,它实现了高级语言中的按值调用(call-by-value)语义。正规序归约首先归约函数作用,这时,参量可能是尚未归约为正规式的表达式。它实现的是按名调用(call-by-name)语义。初期的函数式语言及其执行机制主要采用作用序归约语义,如 MaCarthy 的 LISP 解释器和 Landin 的 SECD 抽象机(SECD-abstract machine)。按值调用语义的重要优点在于参量在使用前计算,然后传给多个使用这个参量的表达式,这样参量只计算一次。但是,对于非严格函数(non-strict function),即函数体不一定需要使用所有的参量,按值调用语义就可能对不需要的参量进行计算,产生无用计算开销。此外,它的实现比较直接和简单,效率较高。例如面向一阶函数式语言的数据流计算方法就是采用按值调用语义。与此相比较,按名调用语义把尚未计算的参量传给多个使用它的表达式,面临着多次计算同一参量的可能性。由于函数式语言的无副作用性,这些计算将得到同样的结果,这就导致了重复计算(冗余计算)。但它保证了对于不需要的参量不会进行计算,所以没有无用计算开销。为了解决按名调用语义中的冗余计算问题,Henderson 提出了惰性计算(lazy evaluation)的方法。它基本上仍是按名调用语义,但在传参机制上吸收了按值调用的优点,对尚未归约的参量传名。而一旦参量计算出来后,将“记住”结果值,下次使用这个参量时就无需重新计算它了。这种方法称为按需调用(call-by-need)语义。

综上所述,采用正规序语义的函数式语言较之基于作用序语义的语言有更强的能力,因为:(a)它具有良好的停机性,保证了计算过程的安全;(b)它可以避免无用计算开销,提高计算的效率;(c)它可以支持高级的程序设计描述,如无穷数据结构等,应用面广。随着函数式程序设计的发展,人们在实现研究中的重点逐渐集中于正规序归约的执行方法和效率上了。

4. 正规序归约的语用研究

由前所述,归约的执行过程就是对 λ -abstraction 中受囿变量进行 β 替代的过程。归约的语用研究就是探讨如何有效地对变量进行替代。它包括 λ -exp 的表示形式和替代形式两个问题。下面将分析几种典型的方法。

(1) 串归约(string reduction)

在串归约中, $\lambda\text{-exp}$ 表示为字符串的形式, 当对 redex 中的函数体进行 β 替代时, 遍历函数体, 为每一个受围变量拷贝一份参量, 并把它们分别插入各受围变量的位置之中。串归约可描述为:

$$(\lambda v. \langle \lambda\text{-exp} \rangle) \langle \lambda\text{-exp}' \rangle \Rightarrow \langle \lambda\text{-exp} \rangle [\text{COPY}(\langle \lambda\text{-exp}' \rangle) / v]$$

考虑到正规序归约时, 参量可能是任意复杂的尚未归约的表达式, 拷贝参量的开销是很大的。此外多个拷贝的参量表达式在后续的归约中可能都要计算, 这样就会产生冗余的计算开销。这些开销可用下例说明。首先定义两个函数: $fxy = x * x + x * y$ 和 $g = f(2 * 2)$ 。要求计算 $(g \ 3) + (g \ 4)$ 。这个程序等价的 $\lambda\text{-exp}$ 为:

$$(\lambda g. + (g \ 3)(g \ 4))((\lambda x. (\lambda y. + (* \ x \ x) (* \ x \ y)))(* \ 2 \ 2))$$

例 11.9 串归约上述表达式过程如下:

$$\begin{aligned} & (\lambda g. + (g \ 3)(g \ 4))((\lambda x. (\lambda y. + (* \ x \ x) (* \ x \ y)))(* \ 2 \ 2)) \\ \Rightarrow & + ((\lambda x. (\lambda y. + (* \ x \ x) (* \ x \ y)))(* \ 2 \ 2) \ 3) \\ & ((\lambda x. (\lambda y. + (* \ x \ x) (* \ x \ y)))(* \ 2 \ 2) \ 4) \\ \Rightarrow & + ((\lambda y. + (* \ 2 \ 2) (* \ 2 \ 2))(* \ 2 \ 2) \ y) \ 3) ((\lambda y. + (* \ 2 \ 2) (* \ 2 \ 2)) \\ & (* \ 2 \ 2) \ y) \ 4) \\ \Rightarrow & + (+ (* \ 2 \ 2) (* \ 2 \ 2)) (* \ 2 \ 2) \ 3) (+ (* \ 2 \ 2) (* \ 2 \ 2)) (* \ 2 \ 2) \ 4) \\ \Rightarrow & 60 \end{aligned}$$

表达式中有下划线的部分表示拷贝的参量表达式。从这个例中可以看到 $(* \ 2 \ 2)$ 被重复计算了 6 次。

所以, 从执行效率上看, 串归约有很大的缺陷, 主要是: (a) 复杂的参量表达式拷贝开销很大; (b) 多个拷贝的表达式冗余计算开销很多; (c) 字符串的插入使表达式管理复杂。但另一方面, 串归约方法却是 $\lambda\text{-exp}$ 归约最直接的实现。如果限制参量在 β 替代时为简单的正规式, 就能基本上避免前述三点缺陷。实际上, 作用序归约一般都采用串归约方法。

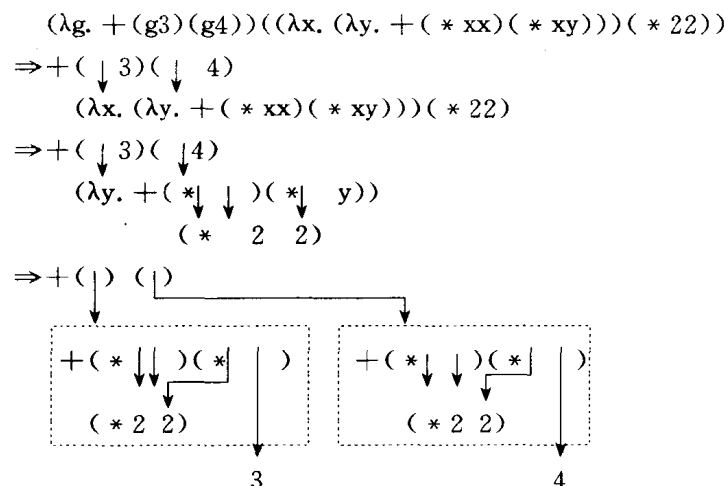
(2) 图归约 (graph reduction)

图归约是一种实现参量共享的方法, 它最早是由 Wadsworth 于 1971 年提出来的。根据这种方法, $\lambda\text{-exp}$ 表示为有向图, 对 redex 归约时, 遍历并拷贝函数体所对应的图, 并把图中的受围变量替代为指向参量表达式所对应图的指针(有向弧), 用有向弧来代表 β 替代的形式。图归约可描述如下:

$$(\lambda v. \langle \lambda\text{-exp} \rangle) \langle \lambda\text{-exp}' \rangle \Rightarrow \text{COPY}(\langle \lambda\text{-exp} \rangle [\text{pointer to } \langle \lambda\text{-exp}' \rangle / v])$$

图归约的一大特性是参量共享。在归约时, 把多处受围变量都替代为指向同一个图(参量图)的有向弧。当第一次计算参量图时, 把归约得到的新图替代原来的旧图, 这时原来指向旧图的有向弧指向的就是新图。以后对这个参量的引用就直接得到归约后的结果图, 没有重复计算。下面给出图归约的一个例子。

例 11.10 用图归约的方法归约例 11.9 中的表达式



其中用虚线划圈部分表示拷贝的图结构部分。从例中可以看到, $(* 2 2)$ 将被计算 2 次。

(3) 全惰性图归约(full lazy graph reduction)

全惰性图归约是一种改进的图归约,其主要特点是减少拷贝函数体的量,即在归约时只拷贝部分函数体,同时共享其它部分函数体。首先分析一下导致拷贝的根源是什么?

考虑到图归约中参量可以被共享,函数(即 λ -abstraction)可以作为参量(高阶函数性质),所以,多个 redex 可以共享一个 λ -abstraction。根据图归约语义, β 替代要破坏 λ -abstraction 中函数体的结构,把其中的受围变量替代为指针,破坏其它共享这个 λ -abstraction 的 redex 的结构。为了避免这种潜在的破坏性,每次 β 替代时,要拷贝函数体,在例 11.10 中可以看到, redex 图(g 3)和 redex 图(g 4)共享 g 弧指向的图 $(\lambda y. +(*xx)(*xy))$ 作为各自的 λ -abstraction。其中弧 x 指向图 $(* 2 2)$ 。(g 3)的归约结果要把 y 替代为指向图“3”的指针,而(g 4)的归约则要把 y 变成指向图“4”的指针。所以需要分别为这两次归约拷贝函数体 $(\lambda y. +(*xx)(*xy))$ 。

上面的分析明确指出,拷贝的原因是 β 替代对函数体的破坏性替代。但是,每一次 β 替代并不是对整个函数体都进行破坏性替代。根据这个想法,可以改进图归约。

再次考察例 11.10 中共享的函数体,它由两分子图组成: $(\lambda y. +(*xx)(*xy))$,其中 x 是有向弧,指向另一个子图 $(* 2 2)$ 。显然,(g 3)或(g 4)归约时,只会破坏含有受围变量表达式,即 $(* xy)$ 的结构,而对 $(* xx)$ 和 $(* 2 2)$ 没有任何破坏。如果拷贝整个函数体,则 $(* 2 2)$ 和 $(* xx)$ 都要拷贝, $(* 2 2)$ 和 $(* xx)$ 将会被计算 2 次。这就是采用基本图归约方法所得到的结果。如果只拷贝 $(* xy)$,那么 $(* 2 2)$ 和 $(* xx)$ 被共享,只计算 1 次。

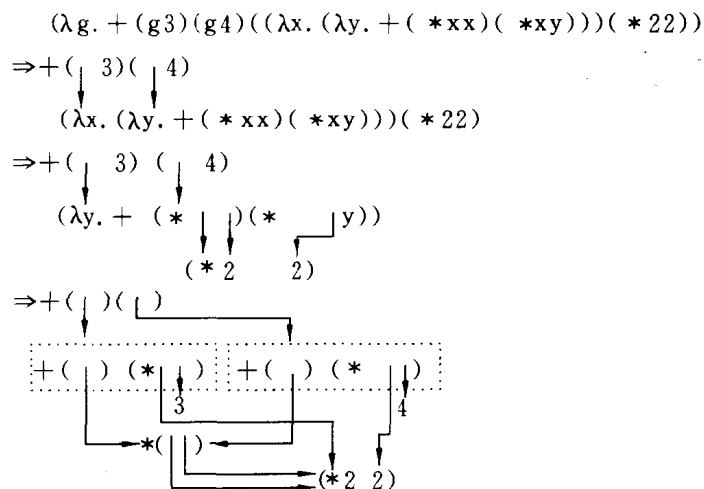
定义 2 给定一个 λ -abstraction, 设为 $\lambda v. \langle \lambda\text{-exp} \rangle$ 。又设 $\langle \lambda\text{-exp} \rangle$ 是 $\langle \lambda\text{-exp} \rangle$ 的一个子表达式。如果 $\langle \lambda\text{-exp}' \rangle$ 的所有自由变量(参见定义 1)中不包括变量 v, 则称 $\langle \lambda\text{-exp}' \rangle$ 是一个自由表达式。如果一个自由表达式不是任何别的自由表达式的子表达式,则称之为是这个 λ -abstraction 中的最大自由表达式 mfe (maximal free expression)。

根据定义 2 可知, mfe 是函数体中最大的与受围变量无关的子表达式,它们在 β 替代

中不受破坏。显然,前面讨论中的 $(* x x)$ 和 $(* 2 2)$ 都是自由表达式, $(* x x)$ 是 mfe,而 $(* 2 2)$ 是 $(* x x)$ 的子表达式。

基于 mfe 的思想,Wadsworth 提出了改进图归约的方法。根据改进的方法, λ -exp 仍用图形式来表示,但在 β 替代时,首先检测出函数体中的 mfe,然后对函数体中非 mfe 的部分进行拷贝,并用参量指针替代受囿变量。这种改进的方法称为全惰性图归约。所谓全惰性(full lazy)性质就是指保证任何参量表达式最多只计算一次。

例 11.11 用 full-lazy 图归约计算例 11.9 中的表达式:



其中划虚线圈的部分是拷贝的图结构。

从上例中可以看到 $(* 2 2)$ 和 $(* x x)$ 的确只计算了 1 次。需要注意的是,对于函数体中的单个标识符组成的子表达式——平凡(trivial)子表达式,诸如原始函数操作符、单个常量或变量,由于它们在执行时没有计算开销,所以不作为 mfe。例如例 11.11 中的“+”操作符没有作为 mfe 被抽取出来共享。

full-lazy 图归约虽然减少了图结构的拷贝,增大了计算的共享程度,但是,它需要花费很大的开销在每次 β 替代时动态地检测函数体的 mfe,因而难以高效实现。

(4) 惰性图归约(lazy graph reduction)

lazy 图归约是在 full-lazy 图归约基础上的进一步改进。它除了保持减少图拷贝的优点外,更主要的特点是采用了有效的检测 mfe 的方法。

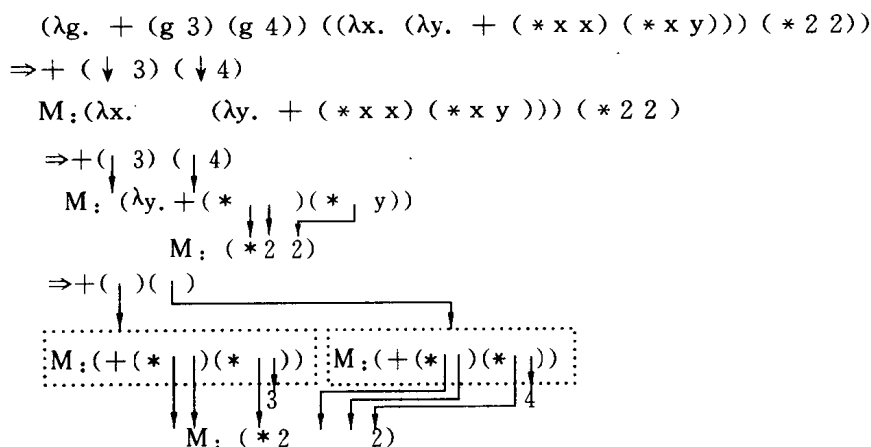
考察一个作用,设为 $(\lambda v. \langle \lambda\text{-exp} \rangle) \langle \lambda\text{-exp}' \rangle$ 。通过 β 替代得到新表达式 $E = \langle \lambda\text{-exp} \rangle [\langle \lambda\text{-exp}' \rangle / v]$ 。显然, E 中可能包含多个子表达式 $\langle \lambda\text{-exp}' \rangle$ 。特别地,对于 $\langle \lambda\text{-exp} \rangle$ 中任一 λ -abstraction, (同时也是 E 中的 λ -abstraction), $(\lambda\text{-exp}')$ 都是一个自由表达式。因为 $\langle \lambda\text{-exp} \rangle$ 中 λ -abstraction 的受囿变量一定不可能在 $\langle \lambda\text{-exp}' \rangle$ 中出现,所以在继续归约 E 时,如果 $\langle \lambda\text{-exp}' \rangle$ 是函数体的子表达式,那么它一定是此函数体的 mfe 的组成部分。这就是 lazy 图归约的基本思想。

在 lazy 图归约中, λ -exp 表示为图形式,图上可以带标记。进行 β 替代时,遍历函数体,对那些从带标记图访问(沿有向弧组成的路径)不到的图结构拷贝,并用指向参量的有

向弧替代各受围变量。这时,给参量图的根结点(root)打上标记。所谓表达式的根结点,就是从它可以访问到此表达式的所有子表达式图的结点。指向参量图的有向弧实际上是指向参量图的根结点。

lazy 图归约具有减少函数体拷贝的特性,但它不像 full-lazy 图归约一样能避免对 mfe 的拷贝。因为参量表达式被替代进函数体后,它只是函数体中 mfe 的一部分,而不 mfe。所以它不能彻底避免重复计算,这也是称之为 lazy 图归约的原因。

例 11.12 对例 11.11 中的表达式用 lazy 图归约计算。



其中划虚线圈的部分是拷贝的函数体结构。“M”表示标记。

从这个例中可以看到 $(*\ 2\ 2)$ 仅计算了 1 次,而 $(*\ x\ x)$ 却计算了 2 次。原因就是只标记了参量表达式 $(*\ 2\ 2)$,而无法标记函数体中的自由表达式 $(*\ x\ x)$ 。

虽然 lazy 图归约不能象 full-lazy 图归约那样彻底避免冗余计算,但它通过一种机械的、简单的检测自由表达式的方法来消除部分冗余计算,在实现效率上要比 full-lazy 图归约高得多。

通过以上的讨论可知,full lazy 图归约语义效果很好,但执行时检测 mfe 的方法不佳。相反,lazy 图归约虽语义效果不如前者,执行时却有简单有效的检测自由表达式的方法。那么是否可以结合它们而得到更理想的图归约方法呢?

首先分析一下造成 full lazy 图归约和 lazy 图归约语义差异的原因。考虑一个 λ -abstraction 的函数体中可以嵌套别的 λ -abstraction,所以外层的 λ -abstraction 的受围变量可能是内层 λ -abstraction 的自由变量。对内层的 λ -abstraction 而言,这个受围变量必是其 mfe 的一个子表达式。当 lazy 图归约进行 β 替代时,被标记的参量表达式替代受围变量,成为内层 λ -abstraction 的 mfe 的一部分。这等价于标记了 mfe 的子表达式。因此,差异的根源在于一个受围变量是否是内层 λ -abstraction 的 mfe 的子表达式。如果受围变量就是内层 λ -abstraction 的 mfe 本身,那么对替代它的参量表达式的标记就是对内层 λ -abstraction 的 mfe 的标记。显然,如果能标记 mfe,lazy 图归约在语义上就同 full-lazy 图归约相同。

(5) 小结

通过对前述几种典型的归约方法的分析比较可以看出,图归约方法是一种较理想的

正规序归约 λ -exp 的方法,它具有简洁统一的表示形式,能很好地支持 lazy 计算语义,较彻底地避免参量的冗余计算。其根本特点在于:

① 克雷化函数语义

所谓克雷化函数(curried function)是指把任何函数操作都看成是只有一个参量的操作。根据 redex 的定义,任何 λ -abstraction 都是只有一个受囿变量的函数操作,只有这样的函数才能进行归约。 β 转换规则则描述了克雷化函数作用到一个参量上时的变换操作方法。当一个 n 元函数作用到 n 个参量上时,则要求归约 n 次。第一次作用到第一个参量上,产生一个新函数,然后把它再作用到第 2 个参量上去,……。从执行的角度看,归约次数越多, β 替代次数越多,开销越大。

② 动态中间函数构造

在克雷化函数归约过程中,必然要产生许多新的函数,它们是动态的,无法预知的。因为同一函数可能作用到不同的参量上去,将产生不同的新函数。这样的动态中间函数不仅要带来大量的构造图开销,而且导致了复杂的图遍历操作。因为当中间函数继续归约时,只能通过遍历才能确定其函数体结构,才能替代受囿变量。

③ 单一的 β 替代规则

β 替代规则是归约的唯一操作方法,它将对计算对象(即函数体)做破坏性修改。为了保证函数体的共享,必须要拷贝函数体。因为,作为唯一的归约规则, β 替代不考虑每次归约的 redex 的具体结构性质,而用拷贝操作来保证一般情况下的正确性。(lazy 图归约方法只是减少拷贝量,而无法避免拷贝)。由于归约过程中产生大量的中间函数,它们的结构是无法预知的,所以,为了拷贝函数体,必须遍历函数体。显然,这样的遍历和拷贝开销是巨大的。

所以,要把图归约方法发展成为实用的执行函数式语言的方法,需要突破 λ -calculus 中以 β 转换规则为核心的归约操作语义,解决图遍历和图拷贝问题。基于这样的动机,Turner 于 1979 年首次提出组合子(combinator)归约语义。

5. 组合子图归约(combinator-based graph reduction)

组合子归约的理论基础是克雷(Curry)的组合逻辑(combinatory logic),由此发展而得的组合子图归约方法是归约计算研究中的一个突破性成果,它在计算对象的描述形式和归约操作规则两个方面改变了传统的 β 归约方法。

(1) 常量作用式 caf

对任何 HT-exp 的 lazy 图归约都可以完全避免参量的重复计算。但是,对于不同形式的 HT-exp,即使它们在内容上是等价的,在执行时的语义也有着深刻的差别。弄清这些差别对 caf 和组合子归约的概念非常重要。下面通过例子进行分析。

例 11.13 对于表达式 $(\lambda x. (\lambda y. + (* x x) (* x y))) (* 2 2)$ 通过不同的转换方法可以得到两个 HT-exp:

$$H_1: (\lambda x. (\lambda p. (\lambda y. + (* p p) (* p y))) x) (* 2 2)$$

$$H_2: (\lambda x. (\lambda p. (\lambda y. + p (* x y))) (* x x)) (* 2 2)$$

它们的归约过程如下(lazy 图归约):

$$\begin{aligned}
 H_1 &\Rightarrow (\lambda p. (\lambda y. + (* pp) (* py))) \downarrow \\
 &\quad M: (* 2 2) \\
 &\Rightarrow \lambda y. + (* \downarrow) (* \downarrow y)) = f_1 \\
 &\quad M: (* \downarrow) \downarrow \\
 &\quad M: (* 2 2) \\
 H_2 &\Rightarrow ((\lambda p. (\lambda y. + p(* y))) (* \downarrow \downarrow)) \\
 &\quad \quad \quad \downarrow \quad \quad \downarrow \\
 &\quad \quad \quad M: (* 2 2) \\
 &\Rightarrow (\lambda y. + (* \downarrow) (* \downarrow y)) = f_2 \\
 &\quad \quad \downarrow \quad \quad \downarrow \\
 &\quad \quad M: (* \downarrow) \downarrow \\
 &\quad \quad M: (* 2 2)
 \end{aligned}$$

从例中可以看到, H_1 和 H_2 的归约产生了等价的结果 f_1 和 f_2 , 但过程却不相同。在 H_1 的归约过程中, 第一步归约没有对 λ -abstraction, 设为 $A_1 = (\lambda p. (\lambda y. + (* p p) (* p y)))$ 产生任何替代。而在 H_2 的归约过程中, 第一步归约就使得 λ -abstraction, 设为 $A_2 = (\lambda p. (\lambda y. + p (* x y)))$ 进行了替代变换。产生这种差异的原因是 A_1 和 A_2 本质上的差别。在 A_1 中, 变量 p 和 y 都是受囿变量 (p 受囿于外层 λ -abstraction, y 受囿于内层 λ -abstraction), 没有任何自由变量。在 A_2 中, p 和 y 也是受囿变量, 但存在自由变量 x 。

定义 3 如果一个 λ -abstraction 中没有任何自由变量, 则称之为闭合 λ 抽象 (closed λ -abstraction), 记为 $c\lambda$ -abstraction。

需要说明的是, 这里所指的一个 λ -abstraction 的受囿变量包括本 λ -abstraction 的受囿变量和与之嵌套 (共享函数体) 的内层或外层 λ -abstraction 的受囿变量。不属于这个概念下的受囿变量的变量才是自由变量。所有嵌套的 λ -abstraction 有共同的函数体, 所以亦有共同的受囿变量和自由变量。在例 11.13 中, 虽然在 H_2 中 exp 是 λ -abstraction:

$$(\lambda x. (\lambda p. (\lambda y. + p(* xy))) (* xx))$$

exp

的子表达式, 但 exp 的函数体是 $(+ p (* x y))$, 而外层的 x 的 λ -abstraction 的函数体是 $(+ p (* x y)) (* x y)$, 所以它们不是嵌套的 λ -abstraction, x 是 exp 中的自由变量。显然 p 和 y 是 exp 的受囿变量。

根据定义 3, $c\lambda$ -abstraction 中的任何标识符要么是常量, 要么是受囿变量。所以, 任何别的 redex 的归约都不会影响这个 $c\lambda$ -abstraction 的函数体结构。在例 11.13 中, A_1 是 $c\lambda$ -abstraction, 因此第一步归约对之没有替代。 A_2 不是 $c\lambda$ -abstraction, 第一步归约对其中的自由变量 x 进行了替代。正是由于 $c\lambda$ -abstraction 的这种“常量”性质, 可以扩充 λ -exp 定义中常量的概念, 把常数、原始函数操作和 $c\lambda$ -abstraction 都定义为常量。由这些常量通过作用 (application) 关系组合而成的 λ -exp 就称为常量作用式 caf (constant applicative form)。例如, “1”、“(+1)”、“(+1 2)”、表达式 A_1 和 H_1 都是 caf 。

caf 是一种特殊的 HT-exp。任一个 λ -exp 都可以用类似转换 HT-exp 的方法转换为 caf 。所不同的是, 在转换 HT-exp 时, 仅要求抽取非平凡的 mfe , 可能留下一些平凡的

mfe,即自由变量。而在转换 caf 时,要求抽取所有的自由变量或 mfe。

cλ-abstraction 对应的函数称为组合子。组合子的一般表示形式为: $(\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E) \dots)))$ 。在 caf 中所有的 λ-abstraction 都是 cλ-abstraction,所以 caf 中的所有函数都是组合子,它是组合子归约中计算对象的描述形式。组合子归约实际上是对一类特殊性质的 λ-exp-caf 的归约。但是,caf 具有同一般 λ-exp 等价的描述能力,这已得到证明。

(2) 归约的重写操作(rewriting operation for reduction)

在 caf 表达式上进行归约操作具有在一般 λ-exp 上归约时所没有的特征,这些特征是产生重写操作语义的根本原因。

定义 4 根据图归约的语义,一个函数作用到参量上进行 β 替代时,需要拷贝的函数体部分称为这个函数的结构(structure)。

函数的结构是一个关键的概念,因为 β 归约方法的主要操作开销就是遍历和拷贝函数的结构。

考虑 caf 中的任一 redex,其形式如下:

$$(\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E) \dots)))E'$$

其中 E 和 E' 都是 caf。通过 β 转换规则可得:

$$C: (\lambda x_2. (\dots (\lambda x_n. E[E'/x_1]) \dots))$$

C 仍是一个组合子。显然,任何别的 redex(只要不以 C 为其函数体)的归约都不会影响 C 的结构。即使归约表达式 E' 中的 redex 也不会。因为根据 lazy 图归约的语义,当 C 作为函数体作用到参量上继续归约时,E' 是无需遍历和拷贝的。另一方面,根据正规序归约原则,E 中的 redex 为能先于以 C 为函数体的 redex 的归约。所以,从产生 C 的这一步归约开始,直到下一步以 C 为函数体的 redex 进行归约之前(即二层相邻嵌套的 λ-abstraction 归约的间隔),C 的结构一直保持不变。这个性质非常重要。假设在产生函数 C 时就以某种方式“记住”C 的结构,那么无论何时何处,当使用 C 作为函数体作用到参量上时,就可以直接“写出”C 的结构,在写的过程中完成受围变量的替代,而不必通过遍历和拷贝来生成归约后 C 的结构。这就是归约的重写操作方法和 β 规则图归约“拷贝+替代”方法的本质区别。组合子归约是对 caf 的归约,它采用的就是重写操作方法。

采用重写操作方法的关键是“记住”生成的函数的结构。caf 表达式的常量性质保证了这样的“记住”是可能的。但由于用 β 规则归约时要产生大量的中间函数,如 C,要动态地记住它们的结构是复杂和困难的。解决这个问题的办法是不产生中间函数。

对于一个组合子,称它定义中最大的嵌套 λ-abstraction 的数目为其元数(arity)。当一个 n 元组合子作用到 n 个参量上时,用 β 规则进行归约的过程为:

$$\begin{aligned} & (\lambda x_1. (\lambda x_2 (\dots \lambda x_n. E_0) \dots)) E_1 E_2 \dots E_n \\ \Rightarrow & (\lambda x_2. (\dots (\lambda x_n. E_0[E_1/x_1]) \dots)) E_2 \dots E_n \\ \Rightarrow & (\dots (\lambda x_n. E_0[E_1/x_1]) [E_2/x_2] \dots) E_3 \dots E_n \\ \Rightarrow & \dots \\ \Rightarrow & (\lambda x_n. E_0[E_1/x_1] [E_2/x_2] \dots [E_{n-1}/x_{n-1}]) E_n \\ \Rightarrow & (E_0[E_1/x_1] [E_2/x_2] \dots [E_n/x_n]) \end{aligned}$$

其中 $E_0, E_1, E_2, \dots, E_n$ 都是 caf, E_0 不是 λ-abstraction。

由于各替代操作 $[E_1/x_1], \dots, [E_n/x_n]$ 相互没有影响,无论以什么顺序替代都会得到同样的结果。所以,为了避免一次替代一个 $[E_i/x_i]$ 而产生的中间函数,在组合子归约中,要求 n 元函数只有当作用到 n 个参量上时,才进行归约。当 n 元函数作用到少于 n 个的参量上时,不进行任何变换。这样的归约语义可描述为:

$$\begin{aligned}
 & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_0) \dots))) E_1 \\
 \Rightarrow & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_0) \dots))) E_1 \\
 & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_0) \dots))) E_1 E_2 \\
 \Rightarrow & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_0) \dots))) E_1 E_2 \\
 & \dots \\
 & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_0) \dots))) E_1 E_2 \dots E_{n-1} \\
 \Rightarrow & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_0) \dots))) E_1 E_2 \dots E_{n-1} \\
 & (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E_1) \dots))) E_1 E_2 \dots E_n \\
 \Rightarrow & E_0[E_1/x_1, E_2/x_2, \dots, E_n/x_n]
 \end{aligned}$$

所以, n 元函数作用到 n 个参量上时,可一步归约得结果 $E_0[E_1/x_1, E_2/x_2, \dots, E_n/x_n]$,这时各替代操作是同时进行的。

综上所述,组合子图归约有两个主要思想:

(a) 在归约操作时,预知函数体的结构,可直接写出函数体的结构,并在重写过程中完成受围变量替代为参量的操作,从而舍弃了 β 规则归约中通过遍历和拷贝来生成函数体、然后替代受围变量的方法。虽然不可能预知参量的内容,但根据 lazy 图归约语义,在构造归约结果时,无须构造参量图本身,仅使用指向参量图的有向弧。另一方面,虽然重写函数体结构在语义上等价“拷贝”函数体的一个备份,但它是根据预知的内容直接“写出”的,而不是通过遍历拷贝出来的。

(b) 归约 n 元函数时,只有当它作用到 n 个参量上,才进行重写操作构造归约结果,从而避免了中间函数的动态生成,减少了归约变换次数,为实现预知函数结构提供了保证。

怎样才能实现对函数结构的预知呢?

基于组合子的常量性质,可以象原始函数一样给组合子命名。如果组合子 $(\lambda x_1. (\lambda x_2. (\dots \lambda x_n. E) \dots))$ 命名为 f ,即 $f = (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. E) \dots)))$,根据组合子归约语义,显然有:

$$\begin{aligned}
 f x_1 \dots x_n &= (\lambda x_1. (\lambda x_2. (\dots \lambda x_n. E) \dots)) x_1 \dots x_n \\
 &= E[x_1/x_1, \dots, x_n/x_n] \\
 &= E
 \end{aligned}$$

这个等式就是描述预知函数结构的方法,它可解释为:当 n 元函数 f 作用到任意的 n 个参量 a_1, \dots, a_n 上时,其归约结果为 $E[a_1/x_1, \dots, a_n/x_n]$ 。

对于一个给定的 caf 表达式 exp,其中所有的组合子函数,设分别命名为 f_1, \dots, f_m ,都是确定的。用函数名替代相应的 λ -exp 后得 exp' ,这时 exp' 中只有常量、原始的函数和 f_1, \dots, f_m 出现,没有任何 λ -abstraction,可称满足这种性质的表达式为 CAF。同时,对应每个函数 f_i ,有:

$$f_i x_1 \cdots x_{n_i} = E_i \quad (1 \leq i \leq m)$$

其中 E_i 只包括常量、原始函数、函数名 f_1, \dots, f_m 和受围变量 $x_1 \cdots x_{n_i}$, 也没有任何 λ -abstraction。所以当 f_i 作用到 CAF 上时, 归约结果中 E_i 的受围变量替代为参量 CAF 表达式, 归约结果必将是 CAF。当归约 \exp' 时, 任一函数 f_i 作用对象都是 CAF, 归约结果也是 CAF。这就是说, 在归约过程中不会产生除 $f_1 \cdots f_k$ 之外的新函数, 归约过程中所需用到的函数全部可以在归约开始前完全确定, 并以 $f x_1 \cdots x_n = E$ 的形式记录下来。这样的描述形式称为归约规则(reduction rule), 因为正是它们决定了表达式的变换形式和语义。

在 β 规则归约中, β 规则是唯一的归约规则, 它不考虑函数(即 λ -abstraction)的结构(实际上也不可能知道), 以一种统一的方法变换各种函数的作用(即 redex)。相比较而言, 在组合子归约中, 有一组确定的归约规则, 它们分别描述了归约过程中各函数作用变换的方法, 对于不同的计算对象, 选择相应的归约规则进行变换。所以说, 归约规则是组合子归约方法的根本特征。下面将分别讨论两种采用不同性质和风格归约规则的组合子归约方法: SK-组合子归约和超组合子归约。

(3) SK-组合子归约

在早期的理论研究中, Curry 证明, 一切可计算函数都可以用两个最基本的函数组合而成。这两个函数分别称为 S 函数和 K 函数, 它们的定义为:

$$S = \lambda f. (\lambda g. (\lambda x. (f x) (g x)))$$

$$K = \lambda x. (\lambda y. x)$$

显然, S 和 K 都是组合子, 可以得到一个通用的组合子归约系统。它只有二条归约规则:

$$(a) S f g x = (f x) (g x)$$

$$(b) K x y = x$$

这就是 SK-组合子归约方法的基本思想。根据这种方法, 任一表达式在归约前都可以, 也必须变换为等价的 S 组合子、K 组合子和常量、原始函数构成的表达式, 这样的表达式非常复杂。例如, 表达式 $(\lambda x. (\lambda y. + x y))$ 转换为 SK 组合子的表示形式后, 得:

$$(S (S (K S) (S (S (K S) (S (KK) (K +)))) (S (K K) (S K K)))) (K (S K K))$$

1979 年, Turner 首次提出的就是 SK-组合子归约方法。为了简化组合子表示形式, 他又引入了许多新的组合子, 主要有:

$$I x = x$$

$$C x y z = x z y$$

$$B x y z = x (y z)$$

$$Y x = x (Y x)$$

并设计了一个编译器, 把 SASL 语言转换为这些组合子的表示形式, 然后利用这些确定的归约规则进行归约。

SK-组合子归约的优点是: 通用性强, 一组确定的归约规则适用于任何表达式。每一条规则都很简单, 每一步归约都可以高效地实现。但它的缺点也很明显, 计算对象描述复杂, 归约次数极多, 每一步归约完成的计算量很少。

(4) 超组合子归约(super-combinator reduction)

在组合子归约中,归约过程由一系列归约步组成,每一归约步对应一条规则的执行,表示一个组合子对一组参量的一次作用。因此,一步归约可以看成是归约计算的一个原子操作,组合子则代表了计算的粒度(grain)。在前面已经看到象 S 和 K 这样的组合子是很小的计算粒度。小粒度计算使计算的次数增多,计算转换和连接的开销增大。

另一方面,大的计算粒度对应着大的归约规则,这意味着归约时要构造大的函数体表达式。一般而言,一条归约规则可以使用多次,每次使用规则都要进行一些相同的工作。所以,归约规则越大,带来重复计算的可能性就越大。

如何才能产生合适的归约规则呢?基于这样的动机,Hughes 于 1983 年提出了超组合子归约方法。其主要思想是构造较大的归约规则,同时消除重复使用归约规则带来的重复计算。

一般的高级函数式语言程序包括一组用户定义函数和一个初始表达式。超组合子归约时,首先把这些函数变换为组合子,那么这些用户函数定义就可以看作为归约规则,它们是用户定义(根据用户需要计算的特定问题)的最大的归约规则。然后分析各归约规则,把其中的重复计算部分抽取出来。最后形成的归约规则就是具有较大粒度、同时又没有重复计算成分的合适的归约规则。

根据组合子的定义可知,如果一个函数的定义体中没有自由变量,它就是一个组合子。只要能把用户定义函数中的自由变量抽取出来,就能把它转换为组合子。

给定一个组合子定义,即是一个归约规则,如何分析并抽取其中的重复计算成分则是超组合子归约的核心。这个问题分为两个方面:什么样的子表达式是重复计算成分?如何把重复计算抽取为共享计算?

显然,一个表达式的公共子表达式(非平凡的)和非平凡的常量子表达式是重复计算成分。因为,无论作用到什么样的参量上,函数定义体中的各公共子表达式和常量子表达式都将得出同样的结果。对于每一次归约,多个公共子表达式将重复计算出相同的结果;对于同一规则的不同次归约,常量子表达式将重复计算出相同的结果。例如,有规则 $F \ x \ y = (f_1(+ \ x \ y) (f_2(+ \ x \ y) (g \ 1 \ 2)))$, 其中 $(+ \ x \ y)$ 是公共子表达式, $(g \ 1 \ 2)$ 是常量子表达式。F 的任一次归约,设为 $F \ a \ b$, 都将得到 $(f_1(+ab) (f_2(+ab) (g \ 1 \ 2)))$, 显然 $(+ab)$ 被重复计算了 2 次。如果 F 分别归约 $(F \ a_1 \ b_1)$ 和 $F(a_2 \ b_2)$, 则得到: $(f_1(+a_1 \ b_1) (f_2(+a_1 \ b_1) (g \ 1 \ 2)))$ 和 $(f_1(+a_2 \ b_2) (f_2(+a_2 \ b_2) (g \ 1 \ 2)))$, 同样 $(g \ 1 \ 2)$ 被重复计算了 2 次。

上述重复计算情形是在归约规则作用到任何参量上时都要出现的。如果考虑到规则作用的参量的具体情况,还可以发现许多不明显的、但是却是重要的重复计算情形,称为部分作用重复计算(partial application recomputation)。例如,有归约规则: $f \ x \ y \ z = (-z (+ \ x \ y))$ 。如果 f 分别归约 $(f \ a \ b \ c)$ 和 $(f \ a \ b \ d)$, 得: $(-c (+a \ b))$ 和 $(-d (+a \ b))$, 这里, $(+a \ b)$ 被重复计算了 2 次。当一个函数作用到两组部分相同的参量上时,可能导致重复计算,这就是部分作用重复计算。其原因是函数定义体中存在仅与部分参量相关的子表达式。非平凡常量表达式是一种与任何参量都无关的子表达式,公共子表达式是在函数体中有多个与部分参量相关的子表达式,它们都可以归纳为部分作用重复计算成分。

抽取重复计算的基本思想为:把可能重复计算的子表达式变成函数的参量,通过图归约参量共享的性质来实现它们的共享。

考虑组合子的一般表示形式: $(\lambda x_1. (\dots (\lambda x_n. E) \dots))$, E 中与部分参量相关的子表达式必是某一层 λ -abstraction 的自由表达式, 这一层 λ -abstraction 的受围变量不出现在此表达式中。例如, 设 exp 是 E 的子表达式, 它与变量 x_1, x_2 相关, 即 exp 中仅有变量 x_1 和 x_2 出现, 那么 exp 则是 λ -abstraction, 设为: $(\lambda x_k. (\lambda x_{k+1}. (\dots \lambda x_n. E) \dots))$, $2 \leq k \leq n$, 的自由表达式。由于不能预知作用的参量情况, 为了保证消除任何可能的部分作用重复计算, 必须假设 E 中任何与任意参量相关的子表达式都会带来重复计算, 都需抽取。抽取算法可概括为:

1° 查找组合子定义体中最内层的 λ -abstraction, 设为 $L = (\lambda v. \text{exp})$;

2° 检测出 L 中所有的 mfe, 设为 e_1, e_2, \dots, e_n ;

3° 定义一个名为 F 的新组合子为:

$$f = (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. (\lambda v. \text{exp} [x_1/e_1, \dots, x_n/e_n]) \dots)))$$

其中 x_1, \dots, x_n 为不出现在 e_1, \dots, e_n 中的新变量;

4° 把程序中所有的 L 替代为 $F e_1 \dots e_n$;

5° 重复 1°—4°, 直到 1° 失败为止。

通过一定的优化, 根据这样的方法产生的组合子称为超组合子, 它保证了 full-lazy 性质, 可以避免任何可能的重复计算。

例 11.14 对子程序:

```
f x y z = (-z (+x y))
start = +(f a b c) (f a b d)
```

其中第一条是普通归约规则; 第二条是初始归约规则, 表示待计算的表达式, 任何程序的第一步归约都是执行初始归约规则。前面分析过, 这个程序直接执行要产生重复计算。现把它们表示为 λ 表达式形式, 按前述算法转换如下:

```
f = (\lambda x. (\lambda y. (\lambda z. (-z (+x y)))))
start = (+ (f a b c) (f a b d))
```

(a) $(x y)$ 是最内层 λ -abstraction, $\lambda z. (-z (+x y))$ 的 mfe, 得组合子 $F_1 = \lambda u. (\lambda z. (-z u))$ 。start 不变, f 变为: $f = \lambda x. (\lambda y. (F_1 (+x y)))$;

(b) x 是最内层 λ -abstraction, $(\lambda y. (F_1 (+x y)))$ 的 mfe, 得组合子 $F_2 = \lambda v. (\lambda y. (F_1 (+v y)))$, start 不变, f 变为: $f = \lambda x. (F_2 x)$;

(c) λ -abstraction, $\lambda x. (F_2 x)$ 中无 mfe, 得组合子 $F_3 = (\lambda x. (F_2 x))$, $f = F_3$, start = $(+(F_3 a b c) (F_3 a b d))$;

(d) $(F_3 a b)$ 是公共子表达式, 可以被看成是非 λ -abstraction 的 mfe, 得组合子 $F_4 = \lambda w. (+ (w c) (w d))$, start 变为: $\text{start} = F_4 (F_3 a b)$ 。

这样得到的组合子归约规则表示形式为:

```
F1 u z = (-z u)
F2 v y = F1 (+v y)
F3 x = (F2 x)
F4 w = (+ (w c) (w d))
f = F3
```

$$\text{start} = F_4(F_3 a b)$$

显然, F_3, f 和 F_2 是等价的, 通过优化, 去掉多余的归约规则, 可得:

$$F_1 u z = (-z u)$$

$$F_2 v y = F_1(+v y)$$

$$F_4 w = (+ (w c) (w d))$$

$$\text{start} = F_4(F_2 a b)$$

这个程序的图归约过程如下:

$$\begin{aligned} \text{start} &\Rightarrow F_4(F_2 a b) \\ &\Rightarrow (+ (\downarrow c) (\downarrow d)) \\ &\quad (F_2 ab) \\ &\Rightarrow (+ (\downarrow c) (\downarrow d)) \\ &\quad F_1(+ \downarrow \downarrow) \\ &\quad \quad a \quad b \\ &\Rightarrow (+ (- \downarrow c) (- \downarrow d)) \\ &\quad \quad (+ \downarrow \downarrow) \\ &\quad \quad \quad a \quad b \end{aligned}$$

从例中可以看到 $(+ a b)$ 是共享的。

超组合子的生成过程是不断分析用户定义函数中的 mfe 并抽取 mfe 生成新组合子的过程, 它实质上是把一个用户定义函数分解为多个新组合子。所以, 在实际应用中, 需要使用多种优化方法来保证避免生成无用的、过多的新组合子。需要指出的是, 这个生成过程是静态的, 不同于 full-lazy 图归约中动态检测 mfe 的开销。

与 SK 组合子归约相比, 超组合子归约规则大得多, 每一步归约操作量大, 保证没有重复计算, 它能以比 SK 组合子图归约少得多的归约步完成计算。但是, 超组合子归约系统中的归约规则是不确定的, 不同的程序定义了不同的归约规则。

(5) 小结

任何可计算函数都可以转换为组合子。组合子是一种特殊性质的函数, 它具有和原始函数一样的常量特性, 保证无论组合子何时、何地、作用到什么样的参量上时, 都能确定归约的结果。组合子归约操作的本质在于在归约前用不同的归约规则来描述各函数的结构, 归约时可根据此直接重写出归约的结果。

超组合子归约是一种高效的归约执行方法, 主要表现为:

- (a) 在归约变换过程中没有图遍历和拷贝操作开销;
- (b) 归约过程中没有任何无用计算和冗余计算开销;
- (c) 每一步归约的计算粒度大, 减少了计算中归约步数和归约步切换的开销。
- (d) 传参机制简单。根据归约规则, 在生成归约结果的同时, 直接把实参写入结果之中。

(e) 归约的结果与环境无关, 归约规则与作用的参量无关。一个表达式的任一子表达式的归约, 其结果与这个表达式的其它部分无关, 这使得归约操作易于并行实现。

当前,对函数式语言的实现主要是采用超组合子归约方法。超组合子作为一种标准的中间语言,其程序形式为:

$$\begin{cases} f_1 x_1 \cdots x_{n_1} = \exp_1 \\ \dots \\ f_m x_1 \cdots x_{n_m} = \exp_m \\ \text{start} = \exp_0 \end{cases}$$

其中前 m 条规则表示一般超组合子 $f_1 \cdots f_m$ 的归约规则,最后一条是初始归约规则,通过它产生程序待计算的表达式 \exp_0 。支持函数式语言的并行图归约计算系统的研究包括:(a) 函数式高级语言到超组合子表示形式的编译方法;(b) 归约规则的描述形式和性质;(c) 并行图归约方法。

11.3.3 图归约的并行实现方法

我们以可计算函数的抽象表示形式—— λ -exp 为出发点,研究 λ -exp 的归约计算方法。通过分析可知,超组合子图归约是一种有效的计算方法。下面我们将以超组合子图归约方法为基础,研究函数式语言的具体实现技术,特别是并行实现技术。它主要包括程序的描述形式和程序执行过程的控制管理策略。这里所指的程序描述形式是超组合子的表示形式,而不是一般函数式高级语言程序。这个层次的研究又称为计算模型的研究。

与归约计算方法的研究相比,计算模型的研究更具有技术性质。采用不同的实现策略,可以设计出不同的计算模型。各国的研究者们已经提出许多种基于超组合子图归约的计算模型,其中比较典型的有:英国 Imperial College 以 ALICE-CTL(compiler target language)为中间语言的 ALICE(applicative language ideal computing engine)系统;英国 Alvey 计划中以 DACTL(declarative alvey compiler target language)为中间语言的 Flagship 系统;英国 University College London 以 FLIC(Functional Language Intermediate code)为中间语言的 GRIP(graph reduction in parallel)系统;瑞典的以 G-code(graph code)为中间语言的 G-Machine Graph Machine 系统;荷兰的以 Lean 为中间语言的 DPRM(Dutch parallel reduction machine)系统和美国 Yale 大学以串行组合子(serial combinator)为中间语言的 DAPS(distributed applicative processing system)系统等。下面将主要讨论计算模型设计中的一些主要思想和问题。

1. 中间语言

中间语言是函数式语言实现研究和计算模型研究中一个非常重要的思想。

从整个系统实现的角度看,中间语言是高级语言和系统结构的界面。如图 11.65 所示,在中间语言之上,可以对多种风格的高级语言 HLL、高级语言的应用系统和编译系统进行广泛的研究;在中间语言之下,可以面向中间语言对其实现技术和相应的系统结构进行研究。这两方面的工作由于有一个确定的中间语言为界面,可以互不干扰,并行进行。在系统结构的研究

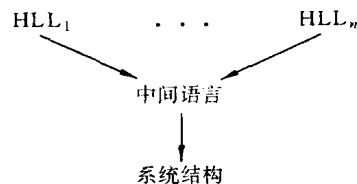


图 11.65 中间语言的地位和作用

中可以不必考虑高级语言中许多面向用户的附加成分。而在高级语言的研究中也不必受实现细节的约束。这样使得各部分的研究更为明确和灵活。

从计算模型的角度看,中间语言是描述计算对象和控制策略的工具。中间语言不仅要描述高级语言中的说明性成分,它还要描述许多执行控制成分,设计计算模型的目标就是设计出一个中间语言,同时以超组合子图归约为核心给出一组机械解释中间语言程序的算法。

需要指出的是,有的中间语言,如 DACTL,还可以描述逻辑程序设计语言的语义。面向这样的中间语言设计系统结构可以同时支持函数式和逻辑式语言。

如前面所述,超组合子的表示可以作为归约系统的中间语言。它包括一组归约规则和一个初始表达式。在实际系统中,一个组合子可以对应多条归约规则,分别表示这个组合子作用在不同参量上的归约结果。由于归约规则在执行过程中将把函数作用重写为归约的结果,所以它又称为重写规则(rewrite rule)。在归约系统中,初始表达式是待计算的对象。当对其中的子表达式—函数作用归约时,根据相应的重写规则产生新的归约结果表达式,并用它重写替代原子表达式,这样计算对象就变成了一个新的表达式。所以说,重写规则所描述的归约结果是要计算的,它们同初始表达式的唯一区别是,在它们之中含有变量。下面首先讨论初始表达式的基本表示形式和性质,对于重写规则可以有类似的结论。

在图归约中,中间语言表示为有向图的形式。计算对象都统一地表示为一张有向图,包括结点(node)、有向弧(arc)和结点的标记(annotation)。有向图的结构用于描述各表达式的相互关系,结点标记用于记录表达式的说明性成分。

在组合子归约中,各表达式的相互关系只有唯一的作用关系。例如,对于 $\langle \text{exp}_1 \rangle$ $\langle \text{exp}_2 \rangle$,说明 $\langle \text{exp}_1 \rangle$ 作为操作, $\langle \text{exp}_2 \rangle$ 作为参量,要求计算 $\langle \text{exp}_1 \rangle$ 作用到 $\langle \text{exp}_2 \rangle$ 上的结果。描述作用关系的图结构有两种:树结构(tree structure)和包结构(packet structure)。在树结构中,所有的操作都被看成是一元操作,为此引入一个统一的结点“@”,它有两条出弧(out-arc),左出弧指向操作,右出弧指向参量,“@”自身代表作用关系。例如,设 f 和 g 都是二元操作,对于表达式 $(f(g\ 2\ 3))(g\ 4\ 5))$,其树结构表示形式如图 11.66 所示。显然, $(f(g\ 2\ 3))(g\ 4\ 5))$ 都被作为一元操作, f 和 g 也被作为一元操作。在包结构中,只有在表达式作为操作的情况下才把操作看成一元操作,因为不能确定操作的元。而对那些

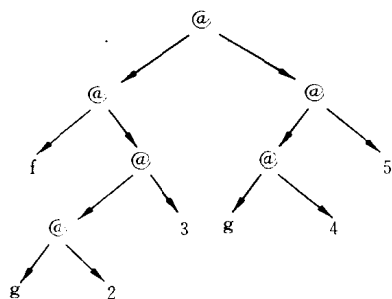


图 11.66 树结构

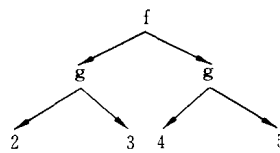


图 11.67 包结构

确定其元数的操作,把结点作为操作自身,用多条出弧指向其参量。出弧的数目不大于结点对应函数的元数。当函数作用到少于其元数数目的参量上时,出弧的数目等于实际作用的参量的数目。当函数作用到多于其元数数目的参量上时,完整的函数作用(一个函数作用到与其元数相同数目的参量上)作为一个表达式操作,它是一个一元操作,可用类似于树结构的方式表示。例如对于前述表达式,包结构表示如图 11.67 所示。

又如,对表达式 $f x_1 \cdots x_n x_{n+1} \cdots x_m$, 设 f 是 n 元操作。这时把 $(f x_1 \cdots x_n)$, $((f x_1 \cdots x_n) x_{n+1})$, \cdots , $((f x_1 \cdots x_n) x_{n+1} \cdots x_{m-1})$ 等都看成是表达式形式的一元操作,可表示为如图 11.68 形式。其中“Apply”是一种特定的函数,语义同树结构中的“@”结点一样。一般而言,包结构的结点数目要比树结构少,结点的表示形式也比较灵活和方便。ALICE、Flagship 和 DAPS 等系统采用的都是包结构。但是,树结构表示的结点较为简单,执行方式,特别是对高阶函数的执行方式比包结构统一和方便。G-Machine 和 GRIP 系统采用的是树结构。本文的讨论将采用包结构的图表示形式。

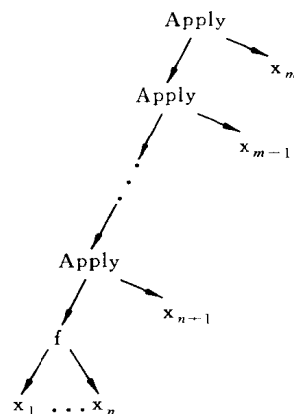


图 11.68 表达式操作的包结构表示形式

中间语言中表达式的说明性成分包括标量常数、数据结构、原始函数和组合子(用户定义函数名)。高级语言中的类型信息可以在编译生成中间语言的过程中处理掉。重写规则中的变量在语义上等同于标量常量。数据结构可由表达式、标量常数和构造符(consructor)组成。例如,表(list)数据结构 $[1, 2, 3]$ 等价于构造符 cons 和标量常数 1, 2, 3 及 NIL 组成的表达式 $\text{cons } 1(\text{cons } 2(\text{cons } 3 \text{ NIL}))$ 。构造符与函数名在结构上有共同的特性,即它们都有确定的元数,表示它们需要的参量数目或子结构数目。cons 就是一个二元构造符。构造符和函数名可统称为操作符(operator)。所以,在中间语言中,说明性成分包括标量常数(简称常数)和操作符。

在图归约中,中间语言程序的计算对象表示为一个有向图,称为计算图(computational graph)。

定义 5 一个有向图如果满足:(a) 它是一个不带任何出弧并标记为常数的结点,或者(b) 它是一个标记为操作符的结点,带有若干指向别的计算图的出弧,则称它为计算图。

计算图是一个连通图,无循环结构。因为函数式语言中循环操作表示为递归函数作用关系。但计算图不是树,因为随着归约的进行,多个表达式可共享参量,一个计算子图可以被多个有向弧所指向。因此,在一个计算图中一定存在唯一的结点,它不被这个图中任何结点所指向,称之为这个图的根结点(root)。所谓一个出弧指向一个图,就是指指向这个图的根结点。如果一个结点有一条出弧指向一个计算图,则称这个计算图是这个结点的参量子图(operand subgraph)。计算图中的结点根据其标记的性质可分为两类:标记为常数或构造符的结点称为构造结点(constructor node),而标记为函数名的结点则称为可重写结点(rewritable node)。以构造结点为根的计算图称为规范图(concanonical graph),以可重写结点为根的计算图称为可重写图(rewritable graph)。

例 11.15 定义一个函数 `Treeheight`, 它用于计算一个非空树的高度。Treeheight 的定义为:

```
data Tree = Leaf(object) # Node(Tree, Object, Tree)
```

```
fun Treeheight(Leaf(object)) = 1
```

```
Treeheight(Node(lt, object, rt)) = 1 + MAX (Treeheight(lt), Treeheight(rt))
```

初始计算表达式为:

`Treeheight(Node(Lefttree, n, Righttree))`, 它对应的初始计算图如图 11.69(a), 其中“+”、“MAX”和“Treeheight”都是函数名, “Leaf”和“Node”是构造符, “1”和“n”是常数。

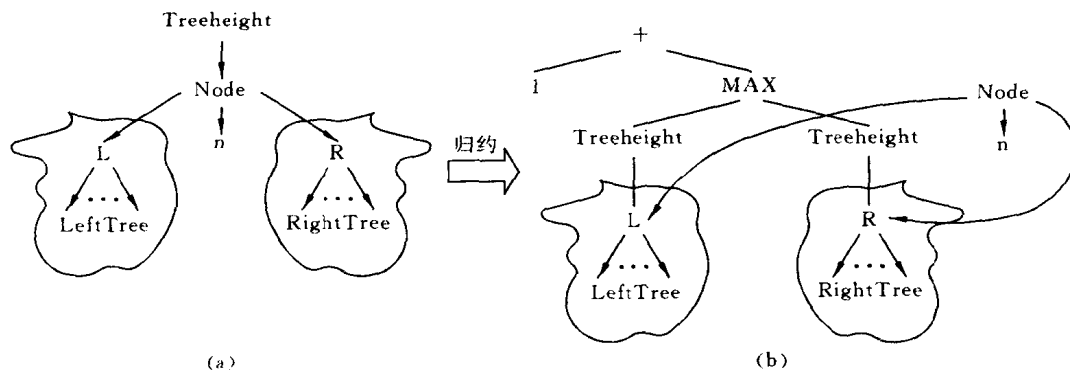


图 11.69 `Treeheight` 程序的计算图

在图归约的中间语言中, 重写规则也表示为计算图的形式。每一规则由模式 (pattern) 和归约体 (contractum) 两部分组成, 形为: pattern:-contractum, 其中 pattern 和 contractum 都表示为计算图, 它说明象 pattern 那样的计算图可以归约变换为 contractum 形式的计算图。例如, 对于例 11.15 中的二条规则, 其表示形式如图 11.70 所示。需要注意的是, 每一条重写规则都对应一个组合子函数, 重写规则中的 pattern 必是可重写图。所以, 只有可重写图才能进行归约变换。在 pattern 中, 根结点的出弧数目定义了函数的元数, 这正符合组合子归约语义中只有当函数作用到与其元数数目相等的参量上时才进行归约的性质, 因为, 只有在这样的条件下, 一个计算对象的图结构才能同 pattern 相匹配。

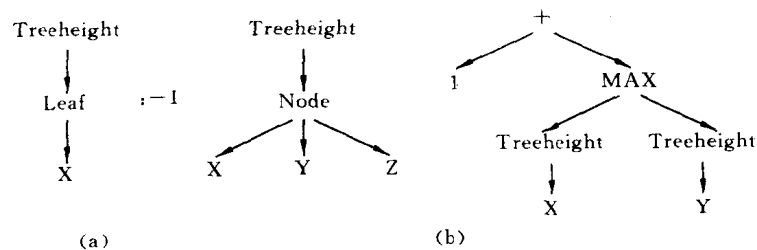


图 11.70 `Treeheight` 的重写规则

在函数式计算中, 计算对象描述了一个唯一的数据结构, 它是这个计算对象的计算结

果。初始计算图是结果的初始表示形式,归约过程中的计算图是结果的中间表示形式,只有最终的规范图才是所需的计算结果表示形式。在整个执行过程中,计算图始终是等价的。图归约的过程就是不断地把可重写图变为规范图的过程,重写规则决定了对什么样的图可以进行怎样的变换。

2. 图变换(graph transmutation)

在图归约中,可重写图就是一个 redex,其中根结点是函数,根结点的出弧所指的子图是函数的参量。这里的 redex 不同于 λ -calculus 中的 redex,它由函数和与其元数数目相等的参量组成。对 redex 图进行变换的结果是用一个新计算图替代 redex 的根结点,如图 11.71 所示。其中 p_i , g_i 和 G 表示子图, f 是 redex 的根结点。这样的图变换操作有二个特性:

(a) 它没有副作用,对 f 的归约计算仅把 f 自身变换为新图 G ,而没有修改其它任何计算对象, $p_1 \cdots p_m$ 和 $g_1 \cdots g_n$ 均保持不变。这个性质使得图变换操作可以在不同的结点上同时执行。

(b) 它符合超组合子归约的 full lazy 语义。 f 是超组合子,它的函数体(即 G)中所有的 mfe 都被抽取到参量中去,所以,图变换时只生成函数体对应的新图 G ,而不重新生成任何参量子图 g_i 。

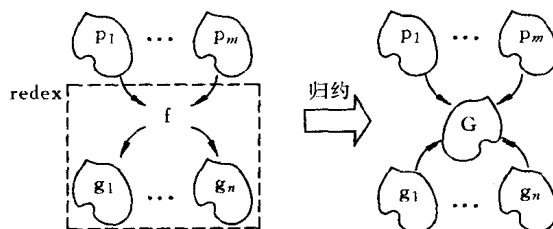


图 11.71 图变换操作

为了实现这样的图变换操作,需要确定如何选择 redex,如何生成新计算图。

形如重写规则中 pattern 的图就是一个 redex。确定 redex 的过程就是计算图和重写规则的 pattern 图进行模式匹配的过程。当一个计算图和一个 pattern 匹配成功,它就是一个 redex,否则,就不是 redex。一般计算图和 pattern 图的唯一差别是 pattern 图中有标记为变量的结点。图的模式匹配过程从根结点开始。首先匹配两个图的根结点,如果根结点匹配成功,则相应的参量子图进行匹配。只有所有的参量子图也匹配成功,这两个图的匹配才算成功,否则失败。结点的匹配原则可概括为:

- (a) 如果两个结点具有相同的标记和相同数目(可能为空)的出弧,则匹配成功;
- (b) 变量结点的和任何结点都匹配成功,并且建立变量名到结点地址的映射关系;
- (c) 当构造结点和可重写结点匹配时,首先归约以可重写结点为根的计算图,直到它变换为规范图后,继续与规范图的根进行匹配;
- (d) 其它情况则匹配失败。

在重写规则的 pattern 图中,根结点为可重写结点,其它结点或是构造结点或是标记为变量的结点。根据上述原则可知,变量结点和可重写结点不影响匹配的进程,与它们匹

配或者马上成功或者马上失败。但是构造结点对匹配过程有很大影响,因为只有它对应的计算图归约为规范图后,才能真正确定匹配成功与否。一个组合子函数的重写规则的 pattern 中所有的构造结点称为这个函数的关键参量(vital operand)。所以,在进行模式匹配选择 redex 之前,需要把计算图根结点函数的关键参量先计算出来。

如果一个计算图和一个 pattern 匹配成功,这就被确定为一个 redex。同时,通过匹配可得到一个映射,设为 $\beta = \{(x_i, n_i) \cdots (x_k, n_k)\}$, 其中 x_i 是变量名, n_i 是结点。选择匹配成功 pattern 所在的重写规则,在计算图空间中生成相应的 contractum 图。在生成过程中,把 contractum 中指向变量 x_i 的有向弧变为指向结点 n_i 的弧。由于组合子函数定义中所有的变量都是受囿变量,所以,对于 contractum 中任一变量 x_i 可以根据 β 找到相应的 n_i 。最后用 contractum 的根替代 redex 的根,完成一次图变换操作。

例 11.16 对于例 11.15 中的初始计算图,它可以和一条重写规则的 pattern 匹配成功,得 $\beta = \{(X, L), (Y, R), (Z, n)\}$, 它是 redex。然后生成相应的以“+”为根的 contractum 图,其中两个 Treeheight 的出弧根据 β 分别指向结点 L 和 R。最后用“+”结点替代原来的根结点“Treeheight”,得到归约结果图。在图变换过程中,原 redex 图中除结点外的所有结点,包括“Node”和“n”结点,子图“Lefttree”和“Righttree”都保留不变。

在图变换过程中,生成的新图要取用新的计算图空间,它只重复使用旧图中根结点的位置,其余旧图结点保持不变。随着旧图根结点被替代,旧图根结点指向其它结点的有向弧也随之消失,而新图不能保证有有向弧指向旧图中各结点,因而存在一些没有任何有向弧指向的结点,它们被继续保留在计算图空间中。这样的结点不再被别的结点引用,称为“废片”(garbage)。为了维护足够的计算图空间供后续的归约使用,必须回收这些废片结点。废片结点的回收(garbage collection)对于实现图归约具有非常实际的意义。

综上所述,给定一个计算图,一般的图归约算法可描述如下:

- 1° 归约计算图根结点的关键参量子图;
- 2° 计算图跟重写规则的 pattern 进行模式匹配
 - (a) 如果计算图跟所有的重写规则的 pattern 都匹配不成功,则归约结束。
 - (b) 计算图跟一个 pattern 匹配成功,产生映射 β ;
- 3° 把相应的 contractum 放入计算图空间中;
- 4° 根据 β 修改新产生的 contractum;
- 5° 用 contractum 的根替代计算图的根;
- 6° 转 1°,继续归约计算图。

这个算法的结束条件是匹配不成功。有两种情形可使计算图与所有的 pattern 都匹配失败。

(a) 计算图的根结点为构造结点,而所有 pattern 的根结点都是可重写结点。这正符图归约把图变换为规范图就结束的语义。

(b) 计算图的根结点虽是可重写结点,但出弧数少于所标记函数的元数,而在所有的 pattern 中,根结点的出弧数目定义了所标记函数元数的数目。这也正符合组合子归约的语义。只有当函数作用到与其元数数目相同的参量上去时,才能进行归约。

从这一节的讨论可知,每一个 redex 的图变换,操作仅仅影响这个 redex 的根,不破

坏别的图结点,这使得可以对多个 redex 同时进行归约,这就是所谓的并行图归约。并行图归约的基本思想是:

(a) 图归约的过程需要执行多个任务(task),每一个任务就是对 redex 的一次图变换操作。

(b) 一个任务的执行将会导致多个任务的执行。最典型的情况是:在通过模式匹配选择 redex 时,需要归约被选计算图根结点的关键参量子图,每一个关键参量子图的归约都对应一个任务。

(c) 多个任务可以并行执行,每一个任务顺序执行。

(d) 任务执行的结果是产生一个新计算图,替代 redex 的根。

基于这样的基本思想,在后面几节中将进一步讨论并行图归约实现中的若干重要问题,包括并行性开发方法(任务的生成方法),并行任务的同步方法和并行计算粒度(任务的大小)。

3. 图归约的并行性开发方法

在图归约中,开发并行性的思想源于归约过程中不断产生新的计算任务这样一种考虑。因此,什么时候、对什么样的计算图(或子图)产生任务是并行性开发策略研究中的关键问题。

对什么样的计算图生成任务决定于选择开发什么样的并行性。在任何计算系统中,并行性可分为保守并行性(conservative parallelism)和投机性并行(speculative parallelism)两类。如果只对整个计算确定需要的子计算成分生成并行计算成分,称为开发保守并行性。这里所谓的“需要”指:如果没有这个子计算的结果,整个计算不能得到结果。如果对所有可计算的成分生成并行计算任务,称为开发投机并行性。开发保守并行性时任一计算都是有效的,都要被使用,因而是安全的,使并行性的控制与管理非常方便。开发投机并行性具有更大的并行计算能力,但它可能执行一些无用计算,甚至是不停机地无穷计算,因而不安全的,使得并行性的控制和管理非常复杂。可以证明,作用序归约是开发投机并行性归约计算的一个特殊情况,而正规序归约语义与开发保守并行性归约计算的语义是等价的。在图归约中开发保守并行性是安全的,而开发投机并行性是不安全的。

什么时候生成任务决定于选择什么样的计算公式(evaluation mode)。计算公式一般分为紧迫计算(eager evaluation)和惰性计算(lazy evaluation)两种。如果一旦一个子计算成分可开始计算,就生成相应的计算任务,称为紧迫计算。如果只有当整个计算需要一个子计算成分计算时,才生成计算任务,则称为惰性计算。这里所谓“需要”指:如果这个子计算不计算出结果,整个计算就不能继续进行下去。对同样的计算量,紧迫计算方式使各任务执行时间重叠大,任务能充分并行执行,而惰性计算公式不能使任务充分并行。在图归约中,可重写结点一旦生成,它就是可计算的,可以生成相应的任务。而只有等到作用到它之上的函数结点开始归约,并且它是其关键参量结点时,这个结点才是需要计算的,必须生成相应的任务。

无论以何种计算方式开发何种并行性,都不是单一的作用序或正规序归约。但是,Church-Roose 定理保证:无论以什么序归约,只要能得到归约结果(停机),这个归约结果就是唯一正确的。所以,在图归约中,只要是开发保守并行性,就能保证并行归约执行的正

确性。

首先分析用惰性计算方式开发保守并行性的方法。

用惰性计算方式开发保守并行性是一种简单和直观的并行图归约计算模型,其执行机制可概括为:

- (a) 对计算图的归约只产生其根结点的任务;
- (b) 在计算图进行模式匹配时,挂起其根结点任务,同时生成根结点的关键参量结点任务;
- (c) 当根结点所有关键参量结点都归约完毕后,唤醒根结点任务继续执行;
- (d) 结点的归约任务执行的结果是产生一个新计算图重写(替代)本结点。重写后,这个结点的任务结束,同时生成新计算图根结点的任务。

在这样的计算模型中,一个任务完成根结点的一次变换。重写后,归约前的根结点变成归约后新计算图的根结点,前者的任务结束,并马上生成后者的任务。经过一系列这样的任务执行,根结点变成构造结点,计算图变成规范图,归约结束。所以,根结点的任务是归约计算图所需要的。当归约根结点时,需要经过模式匹配来选择重写规则,为此必须归约其关键参量结点,所以,根结点的关键参量结点的任务也是需要的。另一方面,只有关键参量结点都归约完毕,根结点任务才能真正进行模式匹配,选择重写规则进行图变换。当完成一次图变换后,要归约重写后的计算图就必须归约新计算图的根结点。所以,所有的任务都是在需要执行时才产生的。

例 11.17 给定一个对搜索树进行插入操作的定义如下:

```
data Search_tree(int)=Leaf(int)# Node((Search)_tree, int, Search_tree)
fun Insert(n, Leaf(n'))=if n>n'
                        then Node(Lode)(n'),n', Insert(n)
                        else Node(Leaf(n), n, Leaf(n'))
Insert(n, Node(lt, n', rt))=if n>n'
                             then Node(lt),n', Insert(n, rt)
                             else Node(Insert(n,lt), n', rt)
```

利用 Insert 函数和 Treeheight 函数计算如下的表达式:

Treeheight(Insert(15, Node(T,5, Node(Leaf(11), 11, Leaf(17)))))

它要求把整数 15 插入一个搜索树中,得到一个新的搜索树,然后计算新树的高度。这个表达式用惰性计算方式开发保守并行性的计算模型执行过程如图 11.72 所示。其中“→”“←”表示所指结点的任务已经生成,并正在执行。“⇒”或“⇐”表示所指结点的任务已生成,但处在挂起等待状态。(为简化图表示,计算图空间中的废片结点没有描述,但它们存在的)。

下面简要解释例 11.17 中图变换的过程:

1° (a)→(b)

在初始计算图中,对根结点生成唯一的任务。执行该任务时发现“Insert”结点是 Treeheight 函数的关键参量结点,所以生成“Insert”任务,根结点 Treeheight 的任务挂起等待。

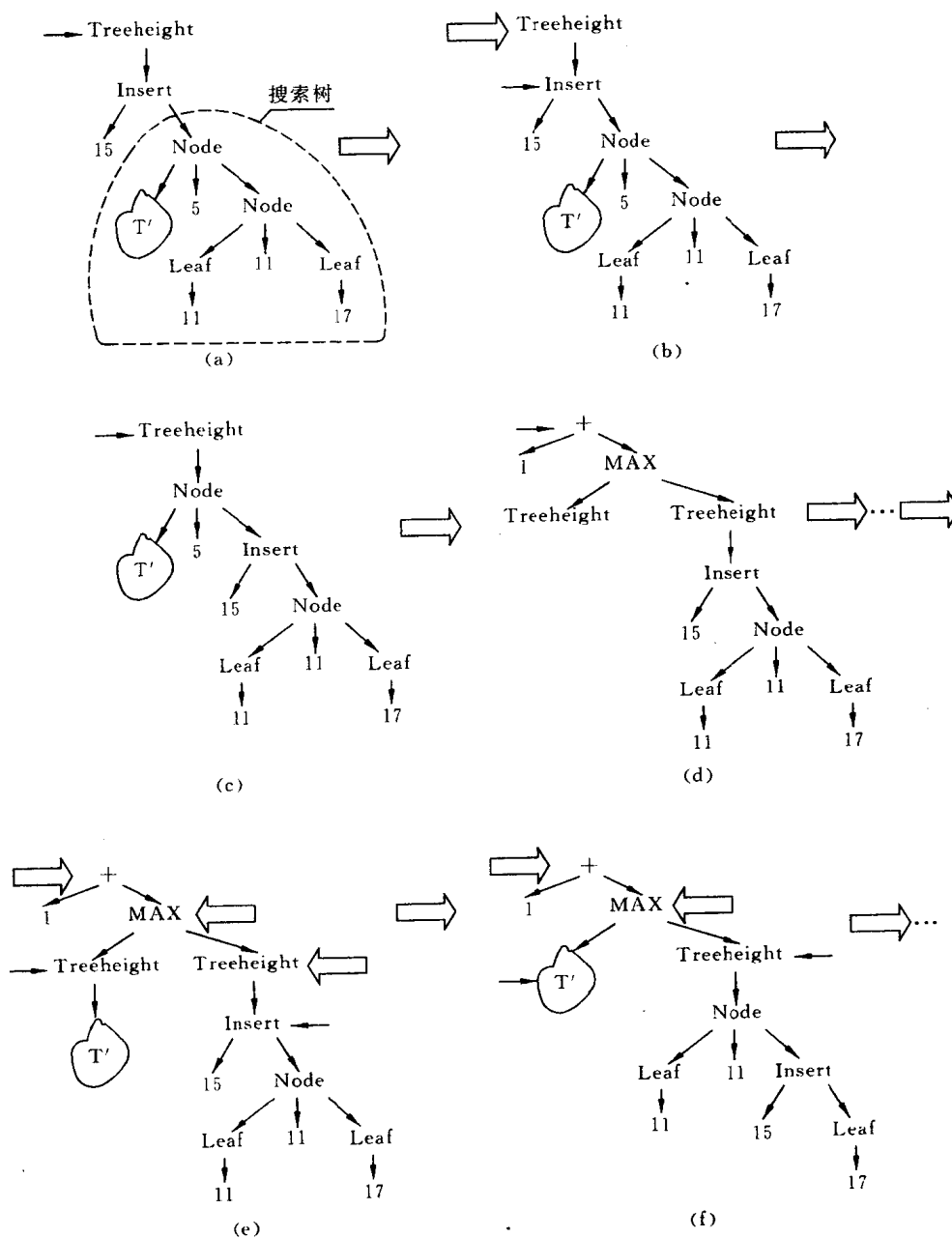


图 11.72 用惰性计算方式开发保守并行性的图归约过程

2° (b)→(c)

执行“Insert”任务。由于 Insert 函数的两个关键参量结点都归约完毕,所以可重写变换生成以“Node”为根的新图重写“Insert”结点。“Node”是构造结点,所以归约完毕,唤醒“Treeheight”结点的任务。

3° (c)→(d)

执行“Treeheight”任务,生成以“+”为根的新图替代“Treeheight”根结点,并生成“+”的任务。

4° (d)→(e)

“+”任务派生“MAX”任务,“MAX”任务派生两个“Treeheight”任务,右分支的“Treeheight”继续派生“Insert”任务。每一个任务一旦派生了子任务,自身总是挂起等待子任务归约结束。所以“+”任务、“MAX”任务和右分支的“Treeheight”任务挂起。左分支的“Treeheight”任务与“Insert”任务无关,可并行执行。

5° (e)→(f)

“Insert”任务执行,产生以“Node”为根的新图。“Insert”任务归约结束,唤醒“Treeheight”任务。

.....

从以上的分析可以看到,基于惰性计算方式开发保守并行性的计算模型执行控制方式是非常简单明了的,易于实现。但其效率却不能令人满意。在这样的模型中,只有参量结点之间才能并行执行(如 MAX 函数的两个参量结点)。而根结点却不能与其参量结点并行归约(如 Insert 任务不能和 Treeheight 任务同时执行)。这正是惰性计算方式的本质所在。只有当一个任务需要另一个结点的归约结果时,才派生相应的任务。显然,派生的任务执行结束前,本任务只能挂起等待。如果一个任务可以派生出当前它不需要的结点的任务,那么它就可以和这些派生的任务同时执行,扩大并行计算能力。用紧迫计算方式开发保守并行性的计算模型的核心,就是一个任务派生它最终一定需要、但当前不需要的计算任务,使得本任务和其派生的任务能并行执行。

在归约一个结点时,确定当前需要的其它结点非常容易,因为各重写规则的 pattern 静态地和显式地描述了各函数的关键参量结点。如果要确定归约一个结点所需要的所有其它结点:当前的和未来的,则比较困难。因为根结点经过一次变换操作后,根结点函数可能要变化。研究用紧迫计算方式开发保守并行性的策略的关键就是解决这个问题。

首先考察函数表达式中的“需要关系”。它实质上是函数操作与其参量的数据相关关系(data dependency)。一个函数与其一个参量是数据相关的,则意味着没有这个参量就无法进行函数的计算。一般的函数可分为严格(strict)和非严格(non-strict)两类。

定义 6 如果一个 n 元函数 f 满足:对任何参量 $a_1, \dots, a_{i-1}, \dots, a_{i+1}, \dots, a_n, (1 \leq i \leq n)$ 有:

$$fa_1 \dots a_{i-1} \perp a_{i+1} \dots a_n = \perp$$

(其中 \perp 表示没有定义(undefined)), 则称函数 f 对其第 i 个参量是严格的。第 i 个参量又称为函数 f 的严格参量。

如果一个 n 元函数对 n 个参量都是严格的,则称为严格函数,否则就是非严格函数。例如,加法函数“+”是一个严格函数,而选择函数 IF, $IF\ x\ y\ z = \text{if } x \text{ then } y \text{ else } z$, 则是一个非严格函数。

函数的严格参量和前面讨论的关键参量是不同的。关键参量只影响函数的一步归约计算,而严格参量将影响整个函数的计算。例如定义函数 $g, g\ x\ y\ z = \text{if } x \text{ then } (y+z) \text{ else } (y-z)$, 它可表示为如下的重写规则形式:

g TRUE $y\ z = y + z$

g FALSE $y\ z = y - z$

在这个函数中, g 的第一个参量是关键参量, 它决定 g 的第一步归约结果为 $y + z$ 或 $y - z$ 。但是 g 的全部参量都是严格参量, 没有 y 或 z 就无法计算出 $(y + z)$ 或 $(y - z)$ 的结果。显然, 函数的关键参量一定是函数的严格参量, 严格参量不一定是关键参量。

函数操作的严格性可以扩展到表达式上。如果一个表达式的计算需要其中一个子表达式的计算结果值, 那么这个子表达式就是整个表达式的一个严格子表达式。

定义 7 给定一个表达式 $(f\ e_1 \cdots e_n)$ 满足:

如果 $e_i = \perp$, $(1 \leq i \leq n)$, 则必有 $(f\ e_1 \cdots e_n) = \perp$

则称 e_i 是表达式 $(f\ e_1 \cdots e_n)$ 的严格子表达式。

由于 $e_i (1 \leq i \leq n)$ 是 f 所代表的操作的参量, 所以又可称 e_i 是严格参量表达式。显然, 函数的严格参量必是这个函数作用表达式的严格参量表达式, 但函数的非严格参量可能成为某个函数作用表达式的严格参量表达式。例如, 对于选择函数 IF, 已知 IF 的第 2 和第 3 参量不是严格参量, 但在表达式 $(IF\ x(y + z)\ z)$ 中, 第 3 个参量表达式是严格的, 而在表达式 $(IF\ x\ y(y + z))$ 中, 第 3 个参量表达式是严格的, 而在表达式 $(IF\ x\ y(y + z))$ 中, 第 2 个参量表达式也是严格的。

更进一步, 严格性的概念可以扩展到数据结构上。根据作用到数据结构上的函数操作的语义, 如果对以一个构造符为根的数据结构操作, 一定需要对其中的一个子结构进行某种操作(可能相同, 亦可能不相同), 可以类似地称这部分子结构为这个构造符的严格参量结构。由于对于同样的数据结构, 在各表达式中可以有多种函数对之进行不同的操作, 所以构造符的严格参量结构不仅由构造符自身的性质, 还需要根据具体的子结构的内容和它们在表达式中被什么样的函数所使用等诸多因素来确定, 它实际上是一种严格参量表达式。例如, 对于前面多次讨论的构造符“Node”, 在表达式 $Treeheight(Node(lt, n, rt))$ 中, 根据 $Treeheight$ 语义, lt 和 rt 都是 Node 的严格参量表达式。而在表达式 $Insert(n', Node(lt, n, rt))$ 中, lt , rt 和 n 都是严格参量表达式。

一般而言, 给定一个计算图, 它对应表达式 $(f\ e_1 \cdots e_n)$, 图的根结点标记为“f”, f 结点的第 i 个参量子图 G_i 对应参量表达式 e_i 。如果 e_i 是这个表达式的严格参量表达式, 则称 G_i 是根结点“f”的严格参量子图 SOS (strict operand subgraph), 如果一个结点是根结点“f”的关键参量结点, 则称以它为根的子图是“f”结点的关键参量子图 VOS (vital operand subgraph)。根据表达式的严格性意义可知, 如果需要归约“f”结点, 一定需要归约“f”结点的 SOS。在用惰性计算方法开发保守并行性的计算模型中, 当归约一个结点时, 只生成其 SOS 任务。而在用紧迫计算方式开发保守并行性的计算模型中, 当归约一个结点时, 生成其 SOS 的任务, SOS 包括 VOS。二者的差别在于那些不是 VOS 的 SOS 的归约任务, 它们与当前本结点的任务可并行执行。

如何分析出表达式中的严格参量表达式是在程序中标记出 SOS 的关键。

1981 年, Mycroft 提出一种基于抽象解释 (abstract interpreter) 的严格性分析技术 (strictness analytical technique), 利用它可以在函数语言程序的编译过程中自动分析函数和表达式的严格性。其后, Clark, Hudak 和 Peyton Jones 等人亦发展了多种严格性自

动分析方法。但是,严格性的自动分析方法目前尚难于实现对程序中所有严格参量表达式的检测。在实际技术中,通常把程序员在程序中的显式标记和编译的严格性自动分析方法结合起来,完成函数式程序中严格参量表达式的分析和确认。

例 11.18 对于前面讨论的 *Treeheight* 和 *Insert* 函数,经过严格性分析可得如下的定义。

$$\text{Treeheight}(\text{Node}(\text{lt}, n, \text{rt})) = 1 + \text{MAX}(*\text{Treeheight}(*\text{lt}), (\text{Treeheight}(*\text{rt})))$$

$$\begin{aligned} \text{Insert}(n, \text{Node}(\text{lt}, n', \text{rt})) = & \text{if } n > n' \\ & \text{then Node}(\text{lt}, n', \text{lt}, * \text{Insert}(n, \text{rt})) \\ & \text{else Node}(* \text{Insert}(n, \text{lt}), n', \text{rt}) \end{aligned}$$

初始表达式变为:

$$\text{Treeheight}(* \text{Insert}(15, \text{Node}(\text{T}', 5, \text{Node}(\text{Leaf}(11), 11, \text{Leaf}(17))))))$$

其中“*”表示一个表达式是严格参量表达式。用紧迫计算方式开发保守并行性的计算模型执行这个程序的过程如图 11.73 所示。

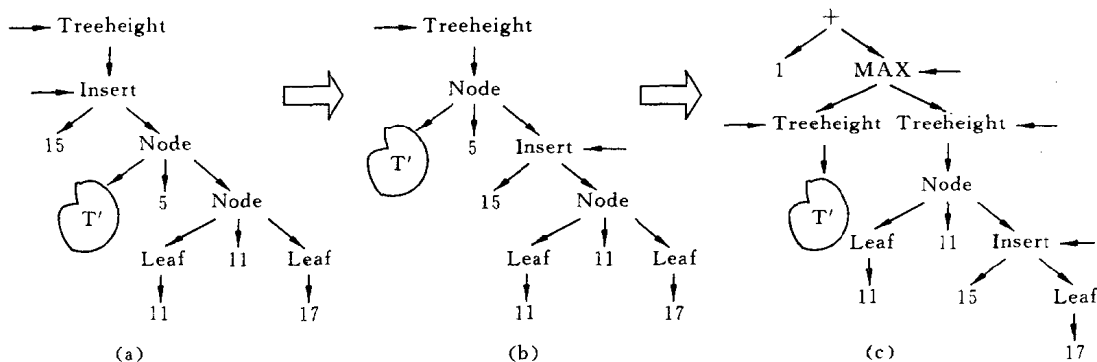


图 11.73 用紧迫计算方式开发保守并行性的图归约过程

图 11.73 中计算图变换过程解释如下:

1° (a)→(b)

在初始归约图中,生成根结点 *Treeheight* 和其 SOS 的根结点 *Insert* 的任务。归约 *Insert* 结点,用以 *Node* 结点为根的新图重写 *Insert* 结点,同时生成 *Node* 结点的 SOS 的根 *Insert* 结点的任务。

2° (b)→(c)

“*Treeheight*”任务和“*Insert*”任务同时执行,生成新计算图,同时对新计算图中根结点的 SOS 的根生成任务。

综上所述,用紧迫计算方式开发保守并行性的计算模型的基本思想可概括为:

- (a) 在中间语言程序中显示标记名 SOS;
- (b) 在归约一个计算图时,生成其根结点的任务,同时归约根结点的 SOS,即生成 SOS 根结点的任务;
- (c) 在执行一个结点的归约任务时,其 VOS 已经开始归约。本任务挂起,等所有 VOS 都归约结束后,唤醒本任务;

(d) 任务执行的结果是生成一个新计算图重写在替代本任务所归约的结点,然后继续归约计算图。

各国对并行图归约计算模型的研究一般都基于这样的计算模型,如 GRIP, DAPS, ALICE 和 Flagship 等系统。这样的模型既具有较高的并行性,又能保证图归约计算的安全性。

4. 并行计算任务的同步(synchronization)

对于并行计算系统,由于并行计算任务之间的数据相关关系和各并行计算任务的执行结束的顺序的非确定性,需要提供同步机制来控制并行计算任务的执行过程。数据相关的并行计算任务可以抽象为生产者(producer)和消费者(consumer)关系。生产者生产出数据送给消费者,消费者根据这些数据确定自身的计算。如果生产者没有计算出结果,消费者就无法进行计算。显然,在图归约中,一个结点的归约任务与其 VOS 的归约任务构成了生产者——消费者关系。

并行任务的同步机制主要包括缓冲方式和同步操作两方面。

并行任务间的缓冲方式与开发并行性的计算方式紧密相关。对于数据驱动计算方式,由于当且仅当生产者产生出计算结果时,马上驱动消费者执行,因而生产者和消费者之间数据无需缓冲,这里采用的是零缓冲(zero-buffer)方式。对于惰性计算方式,首先执行的是消费者,当消费者需要生产者的计算结果时,才开始执行生产者。一旦计算出一个结果,生产者就停止计算。为此,需要在生产者与消费者之间设置一个缓冲区,以它来控制生产者和消费者。如果缓冲区满,则执行消费者,停止生产者。如果缓冲区空,则停止消费者,执行生产者。这里采用的是单缓冲(single-buffer)。对于紧迫计算方式,生产者和消费者同时执行。无论消费者是否需要,生产者只需能计算出来结果,就不停地执行,不断地生成计算结果。为此必须在生产者和消费者之间设置任意多的缓冲区,这样的方式称为无界缓冲(unbounded buffer)方式。无界缓冲方式是保证生产者和消费者并行执行的基本条件。在图归约中,动态变化的计算图描述的是一个数据结构,整个计算图空间就是这个数据结构的缓冲区。VOS 的归约任务(生产者)产生新计算图,通过重写操作把这个新计算图“放入”(替代原图中的结点)整个计算图(缓冲区)中。根结点任务(消费者)通过模式匹配可以访问参量子图,即读取缓冲区内容。所以在用紧迫计算方式开发保守并行性的图归约计算模型中,计算图空间可以自然地实现无界缓冲方式。例如,对于前一节中讨论的“Tree-height”和“Insert”计算程序,“Insert”是生产者,它产生一棵搜索树。“Treeheight”是消费者,它计算搜索树的高度。在一定情况下可以得到图 11.74 那样的计算图。

显然,消费者可以通过 Treeheight 结点的出弧访问缓冲区的头,不断取得所需数据;生产者可能通过重写“Insert”结点访问缓冲区的尾,不断放入计算结果。

并行计算任务同步操作的主要功能是确定任务的执行和停止等待条件。首先分析一个生产者和一个消费者的情况。在图归约的无界缓冲方式中,只有当消费者访问缓冲区的尾部时,表示生产者尚未产生结果,消费者需要等待。因此,结点的标记自然地作为一个同步控制信号。因为在缓冲区中,只有尾部的结点标记为可重写操作符,而其它结点都标记为构造符。例如,在图 11.74 中,缓冲区尾结点是“Insert”结点,其它结点都是“Node”结点。当生产者计算结束后,它用规范图重写缓冲区的尾。规范图的根是构造结点。这时消

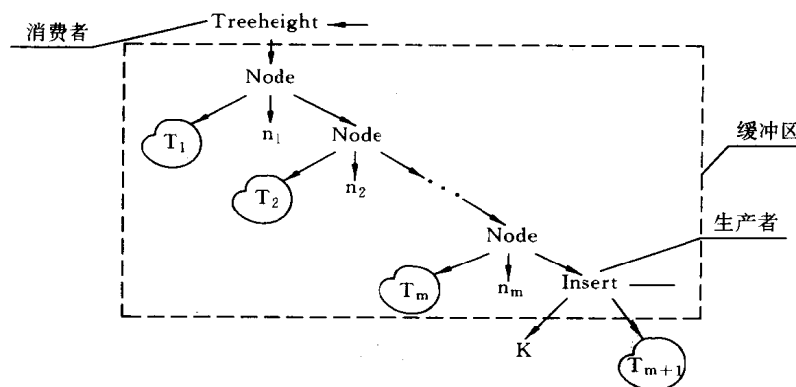


图 11.74 图归约中的无界缓冲区方式

消费者可开始执行,控制消费者停止—执行的方式有两种:忙碌等待(busy waiting)和挂起/唤醒(suspend/wake up)。采用忙碌等待机制时,消费者需要不断地检测缓冲区中结点的标记。采用挂起/唤醒机制时,消费者第一次检查缓冲区中结点,如果发现其标记为可重写操作符,则消费者任务挂起,同时给缓冲区中结点留下一个“标志”(notifier),表示有一个任务正在挂起等待这个结点的归约结果。当生产者归约结束重写这个结点时,可根据这个标志唤醒挂起的消费者开始执行。由于生产者的执行时间是不确定的,往往是较长的,采用忙碌等待方式开销较大。在图归约中,一般都是采用挂起/唤醒方式。

在图归约中,一个结点可以有多个 VOS,而它又可能被多个结点共享。所以,在一般情况下,需要考虑单消费者—多生产者和多消费者—单生产者关系的同步控制方式。对于单消费者—多生产者的情况,消费者任务可能因访问多个缓冲区的尾部而挂起。只有当所有的这些缓冲区都有结果后,才能真正唤醒这个消费者。所以,在结点上需要设置一个需求计数(demand-counter)控制信号(初值为 0)。每当消费者访问一个缓冲区的尾时,除在缓冲区中留下标志外,对自身归约的结点上的需求计数信号量加 1。每一次生产者唤醒消费者时,对消费者对应的结点的需求计数信号量减 1。当需求计数信号量为零时,消费者可以开始执行。对于多消费者—单生产者情况,根据组合归约语义,对每个消费者,这个生产者都将产生出完全相同的结果。如果这个生产者没有产生出结果,多个消费者会分别在缓冲区中共享的结点留下标记。当生产者归约结束时,需要根据这些标记多次唤醒多个消费者。

综上所述,在图归约中并行计算任务的同步控制是通过在计算图结点(缓冲区)上设置信号量和对这些信号量的操作来实现的。它避免了并行任务间直接的控制操作。图归约同步控制机制的主要思想可概括为:

- (a) 采用挂起/唤醒的控制方式;
- (b) 计算图空间被用作无界缓冲区;
- (c) 结点上增加若干标记用作同步信号,包括结点的操作符、需求计数和标志集合等;
- (d) 并行任务的同步过程就是对同步控制信号量的操作。

这样的同步机制已被采用在 DAPS、ALICE 和 Flagship 等系统中。

5. 并行计算任务的粒度(granularity)

在并行计算系统中,任务是并行计算的基本单位,称为粒度。计算粒度太小,将导致并行计算控制管理开销增大,并行任务间的通讯开销增大,诸如生成任务开销和任务同步开销等等。计算粒度太大,又将丧失一个粒度内可并行计算的潜力。因此,寻找合适的并行计算粒度一直是并行计算模型研究中的一个重要问题。

在图归约计算中,并行计算粒度的选择具有很大的灵活性,其根本原因在于组合子的构成是非常灵活的。任意组合子的合法组合都定义了一个新组合子。这意味着可以任意改变定义一个组合子所需要的组合子数目。如果需要把一个组合子变得复杂,只需把这个组合子定义中的各组合子分解为更多、更小的组合子,增大定义组合子的数目。如果需要把一个组合子变得更简单,只须把它定义中的组合子表达式定义为一个新组合子,减少定义中的组合子数目。

在图归约的并行计算模型中,每一个组合子都对应一个图结点,每一个图结点对应一个可并行执行的计算任务。所以,一个组合子的定义中的组合子数目越多,表示这个组合子的归约可以划分为更多的并行计算成分。反之,它就可以划分为更少的并行计算成分。由于每一个组合子的计算量都是确定不变的,划分的并行计算成分越多,表示粒度越小,划分的并行计算成分越少,表示粒度越大。

所以,设计合适的并行计算粒度的问题就是确定合适的组合子定义。

考虑任一组合子的定义,它可能包括用户函数和原始函数。显然,用户函数是需要、也是值得并行执行的,因为用户函数作用往往需要复杂的归约变换才能得到结果。而原始函数作用虽然可以并行,但不值得并行,因为每一个原始函数操作都是非常简单的。类似地考虑,由原始函数组成的表达式也被认为是不值得并行执行的。在组合子定义中,把不值得并行执行的组合子表达式抽取为一个新的组合子。在归约这个新组合子时,不产生计算图,而直接以顺序的方式计算出相应的值。

例 11.19 在“Treeheight”定义中,“+”和“MAX”都是原始函数,为了避免对“+”和“MAX”构造结点并生成任务,“Treeheight”的第二条重写规则可变换为:

$$\begin{aligned}\text{Treeheight}(\text{Node}(\text{lt}, n, \text{rt})) &= \text{New}(\text{Treeheight}(\text{lt}), \text{Treeheight}(\text{rt})) \\ \text{New}(x, y) &= 1 + \text{MAX}(x, y)\end{aligned}$$

其中“New”就是由不值得并行计算的原始函数表达式 $(1 + \text{MAX}(x, y))$ 定义的新组合子。

利用变换后的规则进行图变换的过程如图 11.75 所示。

从图 11.75(a)到(b)的变换可以看出,一个 Treeheight 归约的计算量被分为三个并行计算成分。而在粒度变换前,却是被分为四个并行计算成分。图 11.75(c)到(d)的变换说明,在归约新组合子 New 时,并没有生成表达式 $(1 + \text{MAX}(N1, N2))$ 的计算图,而是直接计算出其结果 R。

综上所述,并行图归约中优化并行计算粒度的主要思想可概括为:

(a) 可以并行计算的组合子对应计算图中的一个结点,每个结点都可以生成并行

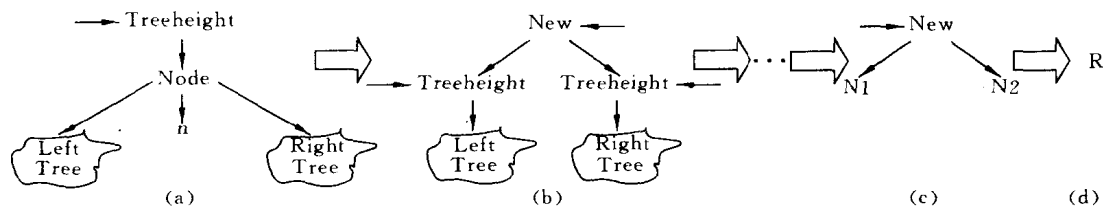


图 11.75 “Treeheight”优化粒度后的归约过程

任务。

(b) 通过抽取新的组合子可以改变组合子的定义体,因而改变一个归约任务中并行计算成分的划分。

(c) 组合子有串行和并行两种结构。对于并行组合子的归约将生成它定义体的计算图;对于串行组合子的归约将不生成任何新的计算图,直接按定义生成计算结果。

(d) 作为组合子的原始函数是不值得并行执行的,因而无需为它生成结点。

通过优化组合子的粒度,可以减少图归约过程中计算图结点的数目和任务的数目,因而减少了计算图结点和任务的生成和管理开销,提高了归约效率。

11.3.4 并行图归约机系统结构

由于函数式语言程序的语义和其并行执行方法同传统的 Von Neumann 计算机有着本质上的差距,因而在传统计算机上难于高效地执行函数式语言程序。函数式语言实现研究的最终目的是设计支持面向函数式语言高效实现的计算机系统结构。在这样的系统结构中,采用许多完全崭新的概念和风格,所以又称为“新奇系统结构”(novel architecture)。

计算机系统结构研究更具有“技巧”(art)性质,它的制约因素非常之多,对一个问题很难有唯一的、显然最优的解决方法。即使在并行图归约机的研究中,虽然同是采用了图归约计算方法,但由于计算模型的具体执行机制不同,最终体现在系统结构的设计上不同。当前世界上对并行图归约机系统结构的研究处于初步发展阶段,在众多的研究计划中,只有 ALICE 和 GRIP 进行了系统结构的详细设计和实现。

一般而言,设计并行图归约机系统结构的关键是要为并行图归约计算模型中最常用的操作提供有效的支持。根据这一出发点,在这一部分中,我们将首先分析并行图归约程序和并行图归约计算模型的特性,然后据此对一般图归约机设计中需要考虑的问题,包括全局机器组织、处理器和存储器组织等进行讨论。

1. 并行图归约计算的操作特性

在并行图归约中,图结点、任务和消息是三个基本成分。

计算图结点是计算的最基本对象,它除了操作符和操作数(实际上是指向操作数地址的指针)外,还包括许多控制符号。从系统结构的角度看,图结点作为计算操作的基本对象,称为包(packet),这是一种带标记的数据(tagged data)。带标记数据表示较之一般数据表示更为复杂。ALICE 系统和 Flagship 系统结构中的包格式如图 11.76 所示。这样的包表示实际上组成了一个数据结构。所以,在图归约计算过程中,数据的基本操作不再是

简单的读/写操作,而是较为复杂的数据结构操作,诸如测试一读(test-read)和测试一写(test-write)操作。例如,当一个结点的归约任务在进行模式匹配之前,需要访问其 VOS 的根。这时它不是简单地读取 VOS 根结点的内容,而是需要检测 VOS 根结点的状态。如果状态表示结点已归约完毕,则读取该结点。如果状态表示结点正在归约,则在其中写入“标志”。所以,对结点的一般操作称为高级访存操作,它需要执行一系列更基本的指令才能完成操作功能。与一般访存操作相比,高级访存操作的开销更大,访存延迟更长。另一方面,在图归约中,图结点的存储方式采用的是堆(heap)的组织形式,这给存储空间的管理带来了很大的复杂性,其中最关键的是废片回收(garbage collection)。图归约过程中必然产生大量废片结点,为了高效地使用存储空间,必须及时地回收废片结点。所以,废片结点的回收操作是图归约中又一重要操作。当前一般采用的方法是基于引用计数(reference-counting)的回收方法。采用这种方法的主要考虑是充分利用计算图中有向弧表示了对结点的引用这个特性。在生成一个新结点时,将产生这个结点指向别的结点的有向弧,表示对那个结点的引用数目加 1。在重写一个结点时,结点中旧的有向出弧被新的有向出弧替代,表示对一个结点的引用数目减 1,而对另一个结点引用加 1。引用数为 0 的结点就是废片结点。这样,废片结点的检测和回收操作可以自然地分布实现,可以和图结点的归约操作并行执行。

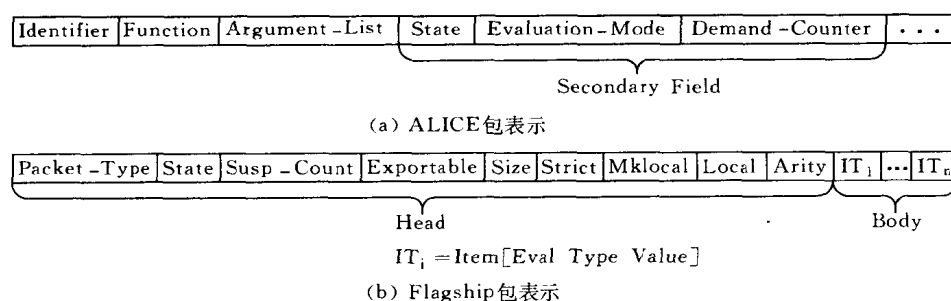


图 11.76 二种并行图归约系统中的包表示

任务是结点的一步归约过程,它对应一条重写规则的执行。考察任务的执行行为需要分析任务的上下文(context)和任务的生命期(life cycle)。所谓任务的上下文是指任务执行过程中需要访问和修改的存储内容。在传统计算机系统中,上下文是一段复杂的程序代码,不同的任务(同时执行)不允许对应同一代码段。程序代码不仅描述了任务执行的操作动作,而且描述了任务需要访问和修改的变量单元。在图归约计算中,一个结点自身就是归约此结点任务的上下文。结点的操作符标记指示了归约这个结点所要进行的操作动作,即重写规则。需要注意的是,重写规则是“纯码”,它只能被任务读取使用,而不能被任何任务修改。所以,同时执行的不同的任务可以共享一重写规则。结点的出弧指示了归约操作的操作对象。在执行过程中,无论出弧所指的结点的具体情况如何,任务都将发出相同的操作。当然,都是高级访存操作。例如,对于一个出弧所指的 VOS,无论这个子图是否归约结束,本任务都将根据重写规则对它执行一个“request”操作。“request”操作是一个高级访问访存操作,它要求取得 VOS 的归约结果。只有当 VOS 的根结点被归约为构造结点

后,“request”操作才算执行完毕。每一归约任务的执行结果都是产生一新计算图,并用新计算图的根结点重写所归约的结点。所以说,一个结点的归约任务仅修改这个结点自身。另一方面,图归约中的任务具有较长的、不确定的生命期,它需要一系列等待/执行周期。任务的执行过程包括三个状态的转换:(a)等待状态。任务执行高级访存操作时,需要等待高级访存操作结束之后才能继续执行后续的操作。因为高级访存操作的延迟可能是较长的和不确定的,如读取关键参量子图的归约结果。(b)准备执行状态。对于长时间的、不确定的等待,任务一般释放处理机,挂起等待。当等待的条件满足之后,任务需要重新排队申请处理机。(c)执行状态。任务占用处理机,执行重写规则描述的操作命令。这些操作命令主要包括新结点的生成、重写结点、同步操作(如唤醒其它等待的任务)和引用计数操作等。它们都可以被看成是高级访存操作,因为所有的都是以结点为目标的。例如,唤醒任务的操作是对一个结点的请求计数信号量的减1—测试—唤醒操作。而生成新结点的操作是对一空闲结点的重写操作。所以说,任务执行的过程,主要是执行一系列高级访存操作的过程,正是这些高级访存操作的延迟使得任务需要经历执行—挂起—准备(被唤醒)—(再次)执行的状态变换。幸运的是,任务的上下文就是结点本身,这给高效实现任务状态变换带来许多便利。

根据前述的讨论可知,结点是任务的上下文。为了同时执行不同的任务,结点需要分布在不同的处理机上。另一方面,一个结点的归约任务需要执行若干对别的结点的高级访存操作,为此需要提供处理机间的消息通讯机制,其中每一个消息对应一次高级访存操作。所以在并行图归约系统中,消息具有两个重要特征:频繁的和短的。因为计算图中任何两个结点都可能分布在不同的处理机上,所以每一个对其它结点的高级访存操作都可能需要进行消息通讯。任务中大量的高级访存操作必然导致频繁的消息通讯。在各类高级的访存操作中,只有派生新结点操作需要较长的消息来传送新结点的内容,而一般高级访存操作都是对已生成结点的部分读和部分修改操作,它们对应的消息内容很短。

综上所述,并行图归约计算从系统结构的角度考虑,具有如下的主要特性:

- (a) 复杂的带标记数据(结点)的存储和管理;
- (b) 大量的高级访存操作和较长的高级访存操作延迟;
- (c) 简洁的任务上下文和复杂的任务执行周期(状态转换);
- (d) 频繁的和短的消息通讯。

在进行并行图归约机结构设计时,必须要充分考虑如何高效地支持这些操作特性。

2. 并行图归约机结构

任何并行计算机系统结构研究中的一个主要问题是处理机存储器的组织关系。对于一个可扩展的机器结构而言,要求系统的计算能力随着资源的增加而增大。显然,由于共享存储器访存速度的限制,在共享存储的多机系统中增加处理机资源将很快达到计算能力的极限。在图归约机中,存储器用于存放计算图,各处理机用于执行不同的任务,对这个计算图的不同子图进行变换。虽然从概念上看,这是一种多任务共享一个计算图的关系,但考虑到系统性能的可扩展能力,在图归约机的结构中一般不采用共享存储,而采用分布式的并行存储方式。并行存储系统由多个可并行访问的存储器组成,它们形成一个统一的全局存储空间。计算图中的各结点分布在各个存储器中。

采用并行存储方式的机器结构又可分为两种：远程存储器-处理机结构(remote store-processor structure)和紧耦合存储器-处理机结构(closely coupled store-processor structure),其结构如图 11.77 所示。

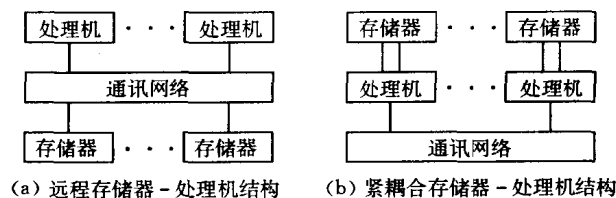


图 11.77 采用并行存储方式的两种机器结构

在远程存储器-处理机结构中,每一个处理机可以执行任一存储器中结点的归约任务,而在处理机执行任务的过程中,每一次的访存操作都要通过通讯网络进行。这种结构的优点是计算图结点的分配和任务的管理较为简单,只要按随机的策略在各存储器上分布图结点,就可以得到最大的带宽(bandwidth)。每个处理机可以从任务存储器中选取可执行的归约任务,保证了各处理机的负载平衡。特别地,这样选取任务的灵活性,给图归约中任务的调度提供了便利。例如,当一个处理机执行一个任务并访问其 VOS 时,发现它尚未归约完毕,本任务只能等待这个 VOS 归约完后才能继续执行。为了使得本任务能尽早执行,需要立即执行其 VOS 的归约任务(如果这些 VOS 尚未被别的处理机执行)。这时,这个处理机可以不选取其它任何任务,而直接取得 VOS 的归约任务执行。无论这个 VOS 分布在哪个存储器中,这样的选取都是可以实现的。另一方面,在远程存储器-处理机机器结构中,通讯网络作为处理机和存储器的接口使得访存延迟较大,这是一个主要缺点。当然,这可以使用一些传统计算机中发展的避免处理机访存瓶颈的技术,诸如 caches 来加以改进。但是,更重要是,可以利用图归约计算模型中任务的无序特性来保证处理机的效率。当一个任务正在访存时,处理机可以迅速切换执行另外的任务。

ALICE 机器和 GRIP 机器结构都是采用的远程存储器-处理机结构。如图 11.78 所示。在 ALICE 机器中,处理机 PA (processing agent) 和存储器 PPS (packet pool segment) 分别与两个通讯网络相联,负载共享系统 LSS 用于传输结点(包),当 PA 执行任务生成新结点时,通过 LSS 把新结点写入 PPS 中。互连网络 IN 用于传送一般消息,它们较之结点消息要频繁,但却要短得多。在 GRIP 机器中,处理器 PE 和智能存储器 IMU (intelligent memory unit) 与一个高速总线相联,所有的消息都通过总线传递。

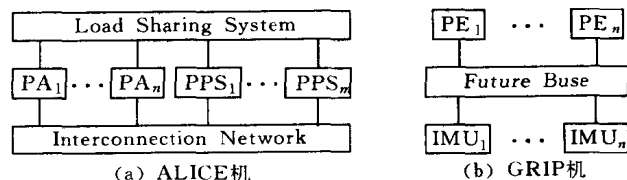


图 11.78 两种图归约机器结构

在紧耦合存储器-处理机结构中,每一个处理机只能执行局部存储器中结点的归约任

务,同时,处理机在执行任务过程中可以通过通讯网络访问别的、远程的存储器。显然,访问局部存储器的速度要比访问远程存储器的速度快得多。采用这种机器结构的主要优点是充分利用计算图的局部性(locality),使得任务执行过程中尽可能多地访问局部存储器,减少远程访存操作;在图归约中,可以采用一系列数据复制(data replication)和优化粒度(optimal granularity)技术来提高计算的局部性。所谓优化粒度是指把多个计算图结点合并为一个计算图结点,把任务对参量结点的访问转换为对任务所归约结点的访问。由于任务所归约的结点一定在局部存储器中,而任务所归约结点的参量结点则可能分布在远程存储器中,这样可以避免某些远程访存操作。所谓数据复制是指在不同的存储器中存储数据的多个备份(或称拷贝),这样使得原来需要远程访问的数据可以在局部存储器中取得。它面临的问题是数据一致性,即当一个数据被修改时,如何保证其多个备份的统一。考虑到图归约计算的特性,可以在多个层次上运用数据复制技术,非常容易保证数据的一致性。首先,可以在各存储器中拷贝所有的重写规则,使得每个处理机在执行任何任务时不必远程读取重写规则,重写规则是“纯码”,任何任务都不会修改重写规则。此外,在计算同步时可采用多种返回机制。当一个结点归约完毕后,它可能需要去唤醒若干别的等待其结果的任务,一般通过对那些等待任务对应的结点的控制标记的减1操作来实现。而被唤醒的任务在模式匹配和生成新计算图的过程中需要访问这个结点的归约值。所以,在唤醒等待任务的同时,可以动态地把本结点拷贝到那些结点的存储器中——称为按值返回(return-by-value),这样被唤醒的任务就不必通过远程访问来读取本结点的内容。由于本结点的归约任务已结束,任何别的结点的归约任务都不会修改本结点,显然,保证了数据的一致性。在这样的机器结构中,最大的困难是如何保持负载平衡。因为这个处理机只能执行其局部存储器中结点的归约任务,即使一个处理机中可执行任务已空,它也不能执行别的存储器中可执行的任务。图归约中的结点对应任务,结点的分布就是任务的分配。在任务执行过程中要动态地产生新计算图,需要把新计算图结点分布到不同的存储器中,以保证各结点的并行归约。这样的动态特性更增加了负载分配的复杂度。另一方面,计算图的分布还要考虑计算的局部性,即结点和引用它的结点应尽量相邻。Keller、Halstead 和 Hudak 等人对此进行了大量的研究,他们的研究表明:必要的结点迁移和任务迁移、动态地统计各存储器的负载量和根据计算图结构确定结点的分配和引用等方法都将有助于保证系统全局的负载平衡。

Flagship 机和 DAPS 系统都是采用的紧耦合存储器-处理机结构,如图 11.79 所示。

3. 并行图归约机的存储系统

存储系统的研究是并行图归约机系统结构研究中的一个关键问题。这是因为在并行图归约计算中,计算图结点是一种复杂的带标记数据表示形式,并且大量的高级访存操作构成了执行过程的主要开销,所以用一般传统存储系统将极大地影响任务执行的效率。

与传统计算机中的存储系统相比较,并行图归约机的存储系统具有更多的“智能”,称为智能存储器(intelligent memory)。它不仅要支持一般的读/写操作,更要能高效地支持高级访存操作。高级访存操作的实现需要通过一系列的读、写、测试和算逻运算来完成。所以,在智能存储系统中,除了提供存储空间外,还需要提供附加的处理机。它用于接收、分析和解释高级访存操作命令,把它们转换为更低级的“微码”,对存储空间进行操作。高级

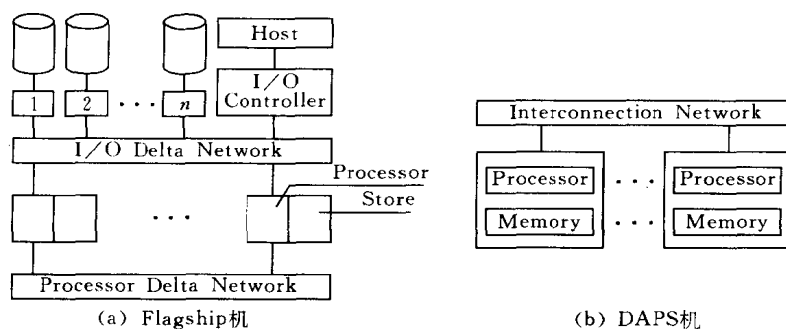


图 11.79 采用紧耦合存储器-处理机结构的图归约机

访存操作以结点为操作对象，“微码”却以结点内部的子结构为操作对象。

采用智能存储系统可以大大减少处理机和存储器之间的通讯量,因为当处理机直接用“微码”访存时,需要执行多条“微码”命令才能完成一条高级访存操作的功能。在远程存储器处理机结构中,对于减少通讯网络的通讯量和网络的延迟无疑是非常有益的。同时,在紧耦合存储器处理机结构中,智能存储器可以直接响应远程处理机对它的访存操作,而不必由局部处理机处理。这对于提高任务执行效率非常重要。实质上,智能存储器可以和处理机同时执行归约任务,它分担了一部分处理工作,使整个系统的效率得到提高。

智能存储系统分担的处理工作一般可分为三类:(a)接收并处理任务发出的高级访存操作,如同步、唤醒和生成图结点等;(b)管理计算图空间,如动态存储分配和废片回收等;(c)管理任务队,诸如任务的生成、删除和状态转换等。这些操作与处理机中进行的归约操作在性质上有所区别。归约操作确定计算图如何进行变换,如何计算出结果;而智能存储系统的处理工作则是进行资源的管理。因此,在智能存储器的研究中的主要问题是如何抽象出归约计算中最普遍的高级访存操作,并基于它们设计有效的资源管理算法和实现技术。

在 ALICE 机中,PPS 是一个智能存储器,其结构如图 11.80 所示。它由两个可编程的微处理器 INMOS Transputer 和 2M 内存空间组成。存储空间分为三个部分:PPP 存放可归约的结点,PEP 是自由空间,DP/SP 存放归约完毕或当前不能归约的结点。由于结点本

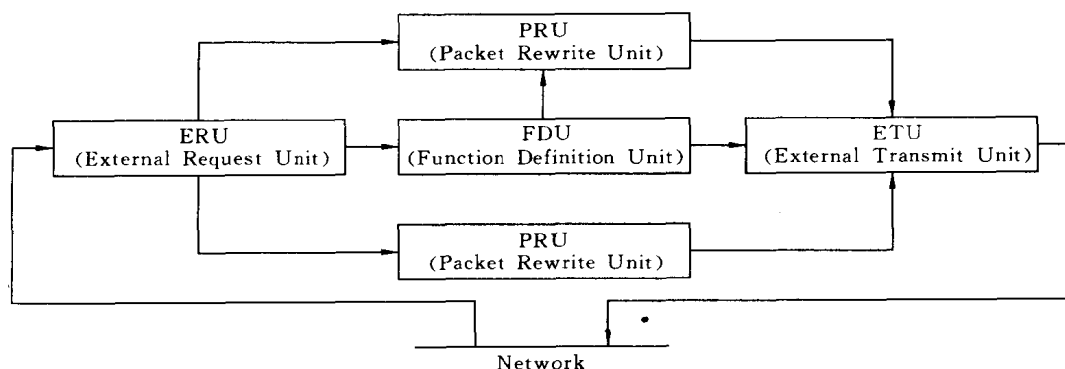


图 11.80 ALICE 智能存储器

身表示了任务的上下文,所以 PPP 实际上构成了待执行状态的任务队,而 SP 构成了挂起任务队。一个 Transputer 用作包管理部件 PMU,它从通讯网络上接收消息,对存储空间进行相应的处理,产生处理结果送给外部传输部件 ETU。ETU 也是一个 Transputer,它负责把 PMU 的处理结果以消息的形式发送到通讯网络上。

4. 并行图归约机的处理机

并行图归约机的处理机系统用于执行归约任务,它主要完成重写规则描述的模式匹配,生成新计算图和重写操作。从前面的讨论可知,在任务执行过程中有大量的高级访存操作,智能存储器需要较长的延迟时间才能完成这些操作。处理机的设计必须保证不因这样的延迟而降低处理机的利用率,即有较好的延迟容忍性(latency tolerance)。另一方面,任务具有较长的和复杂的生命期,其中最关键的是从占用处理机执行到释放处理机挂起等待的状态转换。当一个任务挂起时,处理机应切换执行新的可执行任务。在任务切换时需要保留挂起任务的上下文,同时建立新执行任务的上下文。处理机的设计必须采用高效的上下文切换技术来满足归约任务频繁切换的特性。总而言之,如何实现较好的延迟容忍性和快速上下文切换是并行图归约机中处理机设计的两大基本问题。

解决访存延迟的基本技术是通过指令流水执行方式来减少处理机等待访存应答的空闲时间,但是由于指令间存在的相关性常常会导致流水线“断流”,即前一操作指令若没有完成则后面的指令将无法执行。例如,在生成新计算图的过程中,如果生成一个结点的访存操作(写结点操作)没有完成,就无法得到这个结点的地址,因此,那些需要引用这个结点的其它结点的生成操作就无法执行。在流水执行的基础上发展起来的多线程(multiple thread)控制流思想可以克服这个缺陷。根据这思想,在处理机中同时装入多个任务,以流水的方式交叉执行不同任务的重写规则指令。由于流水线中的指令分属不同的任务,彼此具有独立性,可以避免通常的“断流”现象,保证流水线的饱和。这样的多线程控制流思想在许多新型的并行系统结构中得到应用,如 HEP-1 机、Concurrent Lisp 机和 ALICE 机等。

采用多线程控制流思想设计的处理机带有一个庞大的寄存器堆(register file),它划分为若干寄存器集(register set)。每个寄存器集用于存放一个任务的上下文。因此,一个处理机可以看成是多个虚拟的归约部件 VRU(virtual reduction unit)在同时工作,每一个 VRU 执行一个任务。

解决快速上下文切换问题的基本方法是减少切换的频度,简化上下文组织。仔细考察任务等待的性质可以发现,有的高级访存操作的延迟是相对短的和确定的,如对局部存储器的访问,生成一个新结点操作等。对此任务可用忙碌等待(busy waiting)的方法来不断测试操作的结束条件是否满足。这时无需切换任务。只有对那些相对较长的和不确定的操作,如远程访存操作、读取一个尚未归约结束的值等,才把任务切换出去。由前面一节的分析已知,图归约计算的一个重要特性是任务的上下文直接反应在相应的图结点上,任务执行过程中没有除图结点外的暂时中间结果。因此处理机中上下文组织可以非常简单地表示为相应结点的地址。在任务切换时,显然无需保存这个结点地址,上下文切换开销几乎为零。但是,这样简单的上下文组织使得任务执行过程中访存次数增多。通过把一些任务可能频繁访问的图结点的结构装入处理机的寄存器中构成任务的上下文组织,可以降

低存储器和处理机间接口的带宽。另一方面,考虑到优化并行计算粒度后,几个逻辑上的结点被“包含”在一个任务中,逻辑结点的归约结果形成了任务的中间结果,如果把它们全部组织在处理机中,无疑会增加上下文组织和切换的复杂度。因此,需要用一些系统结点把中间结果组织在图空间中,以保证在不失上下文组织的切换的简单性条件下优化并行计算粒度。

在 ALICE 机中,处理机由 5 个 Transputer 组成,其结构如图 11.81 所示。其中外部请求部件 ERU 和外部传输部件 ETU 分别用于发送和接收通讯网络的消息。函数定义部件 FDU 用于存放重写规则。

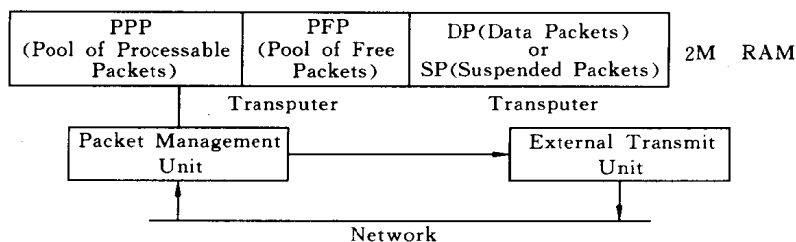


图 11.81 ALICE 处理机结构

两个包重写部件 PRU 用于执行重写规则。每个 PRU 可以同时装入四个任务,相当于四个虚拟包重写部件 VPRU (virtual packet rewrite unit)。在执行过程中,任一空闲 VPRU 向 ETU 发送任务请求命令,从远程的 PPS 中取得一个任务。需要注意的是,由于 ALICE 机采用远程存储器-处理机结构,每一次访存都要经过网络延迟,所以,为了减少访存次数,其任务上下文较大,把整个包(结点)传入处理机作为任务的一下文组织。ERU 把包传给 FDU, FDU 负责取得这个包对应的重写规则。如果本 FDU 中没有所需的重写规则,FDU 可发送命令,通过网络的消息通讯从别的处理机中的 FDU 中读取所需的重写规则。FDU 把包和相应的重写规则送给空闲 VPRU 所在的 PRU。PRU 根据重写规则描述的指令归约结点,其间需要多次通过网络的变换访问 PPS。例如,当 PRU 需要读取结点的参量结点时,向 ETU 发出命令请求,ETU 通过网络把消息发往参量结点所在的 PPS。ERU 从网络上接收 PPS 送回的参量结点包,并把它直接送入 PRU。在这段网络交换处理的延迟过程中,PRU 可以切换执行其中其它 VPRU 的任务。当一个任务结束或挂起时,PRU 把相应的包送往 ETU,通过网络把它写入这个包原来的 PPS 中去。这时 PRU 的一个 VPRU 又成为空闲 VPRU。

5. 小结

前面几节的讨论主要是从并行图归约计算模型的角度出发,对并行图归约机中的操作特性,以及它们对资源组织结构的要求进行了分析,对并行图归约机系统结构的主要设计问题和一些解决方法进行了讨论。这些对于并行图归约机系统结构的设计是必要的,但不是充分的。为了设计通用的、可扩展的、实用的并行图归约机,不仅要研究图归约计算的性质,还需要充分研究和利用并行处理技术和最新的硬件技术,诸如通讯网络、虚拟存储、分布式 I/O 子系统和 Caching 机制等。

另一方面,在并行图归约机系统结构的研究中,各种执行控制策略往往是矛盾的。如

计算图分布的局部性和负载平衡,快速任务切换和任务访存频度等。如何综合考虑各种控制策略和资源组织以使整个系统达到最优的性能,是系统结构设计的一个重要问题。这需要建立系统结构的模拟/仿真系统,建立系统结构的分析模型,对系统结构的各性能指标进行量化分析和研究,诸如,网络的通讯量及延迟、处理机与存储器的比率、一个处理机实现多个虚拟归约部件、整个系统的归约速度及速度单位的定义等等。

习 题 十 一

11.1 解释下列名词:

数据流计算机;控制流计算机;控制驱动方式;数据驱动方式;
需求驱动方式;串行控制流模型;并行控制流模型;单赋值规则;
静态数据流计算机;动态数据流计算机;函数式程序设计语言;
归约机;串归约;图归约机

11.2 分别用串行控制流模型和并行控制流模型计算函数:

$$z = a + b + c + d + e + f + g + h$$

给出指令在指令存储器和操作数在数据存储器中的分布情况。

11.3 分别用结点分支线法和活动片法画出计算函数:

$$x = (a + b)^2 + (a - b)^2$$

的数据流程图。

11.4 对于下面的条件语句:

if true then $z = x + y$ else $z = x - y$

在用结点分支线法画数据流程图时,回答下列问题:

- (1) 需要采用哪几种结点? 每种结点的作用分别是什么?
- (2) 至少需要输入哪几个起始数据令牌和起始控制令牌? 这些令牌分别携带什么样数据或控制信号?
- (3) 用结点分支线法画出数据流程图。

11.5 用结点分支线法画出求一元二次方程两个实数根的数据流程图,基本的运算结点有: +、-、 \times 、 \div 和 $\sqrt{\quad}$ 。

- (1) 画出各个算术操作结点之间的数据相关关系和控制相关关系图,数据相关关系用带箭头的实线表示,控制相关关系用带箭头的虚线表示。
- (2) 指出哪些基本运算操作可以并行执行? 如果每个操作结点的执行时间均为一个时钟周期,则整个计算过程总共需要多少个时钟周期?
- (3) 画出计算过程的数据流程图。

11.6 用结点分支线法实现下面这个简单的 C 语言程序:

```
main( )
{
    int i,x,y,z;
    i=10;
```



```

while(i>0)
{
    if(x>y) z=z+x;
    else z=z+y;
    i--;
}
}

```

- (1) 用结点分支线法画出数据流程图。
- (2) 为了让这个数据流程图开始运行,至少需要哪几个起始数据令牌和起始控制令牌? 这些令牌分别携带什么样的数据或控制信号? 它们的作用各是什么?
- (3) 在数据流程图中有几个并行执行的循环? 每个循环的功能是什么? 各个循环内包括有哪些基本操作?
- (4) 如果每个操作结点的执行时间均为一个时钟周期,采用结点分支线法执行完上述这个程序总共需要多少个时钟周期?

11.7 求两个向量 A 与 B 的和 $C=A+B$,它的 C 语言程序如下:

```

main( )
{
    int i,n;
    float r1,r2,r3,a[n],b[n],c[n];
    n=100;
    for(i=0;i<n;i++)
    {
        r1=a[i];
        r2=b[i];
        r3=r1+r2;
        c[i]=r3;
    }
}

```

- (1) 假设每个基本操作均需要一个时钟周期,如果采用串行方法执行,包括循环控制操作在内总共需要多少个时钟周期?
- (2) 采用软件流水技术重新改写循环部分的程序(不包括循环控制部分),要求有尽可能多的操作并行执行。
- (3) 模拟执行用软件流水技术改写后的程序(至少 4 次循环),写出每次循环执行过程中寄存器 r1 中的内容被赋值和引用的情况,从中发现了什么问题? 请给出至少两种解决这个问题方法,并重新改写这个循环程序。
- (4) 假设每个处理机每个时钟周期只能执行一个基本操作,采用软件流水技术执行这个循环程序(不包括循环控制部分)需要几个处理机? 至少需要多少个时钟周期?
- (5) 用结点分支线法画出这个循环程序的数据流程图。

11.8 对于数据库计算机和知识库计算机,解释下列名词:

软件后端机数据库机；智能控制后端机知识库机；硬件后端机数据库机；
智能接口；过滤器；逻辑接口；物理接口；旋转处理技术

11.9 叙述函数式程序设计语言的主要特征。

11.10 假定函数式程序中有如下定义：

$a=5, b=3, c=(+a\ b), d=(-a\ b), +, -, *$

(a) 若用顺序的需求驱动方式，写出表达式 $(* \ c \ d)$ 的求值过程。

(b) 若用并发的求值方式，写出表达式 $(* \ c \ d)$ 的求值过程。

11.11 分别在图归约机上画出表示

$<+, (<*, (A, B)>, <-,(C, D)>, E)>$

的二叉树和多叉树表示图。

第十二章 实验: DLX 处理器

本章将介绍一个虚拟的处理器:DLX 处理器。该处理器是 Patterson 和 Anderson 在其“Computer Architectue: A Quantitive Approach”一书中提出的。该处理器反映了新一代处理器的特点。通过了解 DLX 处理器的结构和工作原理,并利用 DLX 模拟器进行实验,可以帮助读者综合地了解和运用有关处理器指令系统的设计、流水线的设计与实现等方面的知识,有助于对本书前面章节所述内容的理解。

12.1 DLX 基本结构

在这一小节中我们将描述一种被称为 DLX(读作 Deluxe)虚拟存储器。笔者认为 DLX 是一组在设计思想上与 DLX 相似的实验和商用处理器的代表。这些机器包括 AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260 等。

DLX 的指令集结构具有下面的特征,在本书的前面部分讨论过这些特征。

- (1) 使用 load/store 结构的通用寄存器。
- (2) 支持以下寻址模式:变址(地址空间为 12~16 位)、立即(8~16 位)、寄存器寻址。
- (3) 支持简单的指令系统,因为这些指令占所执行指令的绝大部分,包括:load, store, 加,减,寄存器到寄存器转移,与,移位,测试相等,测试不等,分支(使用至少 8 位的 PC 相对地址),跳转,调用,返回。
- (4) 支持的数据类型:8 位、16 位和 32 位整型,64 位 IEEE754 浮点数。
- (5) 侧重处理器性能时使用固定指令编码,侧重代码大小时使用可变指令编码。
- (6) 提供至少 16 个通用寄存器和单独的浮点寄存器,确保所有的寻址模式都可应用于所有的数据转移指令,并形成最简指令集。

我们将展示 DLX 是怎样实现这些特征的。和许多目前的机器一样,DLX 强调:

- (1) 一套简单的 load/store 的指令集;
- (2) 能够提高流水线效率的设计,包括一套固定的指令编码;
- (3) 使得编译器易于产生高效的目标码。

DLX 是一种适用于学习和研究的处理器结构模型。这种类型的机器正在日趋流行,而且其结构非常易于理解。

12.1.1 寄存器

DLX 有 32 个 32 位的通用寄存器(GPR),名称为 R0,R1,...,R31。还有一套浮点寄存器(FPR),它既可用作 32 个 32 位单精度寄存器也可用作 16 个双精度寄存器。这些 64 位的双精度浮点寄存器被命名为 F0,F2,...,F28,F30。指令系统提供了单精度和双精度

的浮点操作。

R0 的值永远是零。我们将在后面看到是怎样用这个寄存器来从简单指令集成一组有用的操作的。

一些特殊寄存器可以与整数寄存器交换数据,如用来 store 有关浮点操作结果信息的浮点状态寄存器。还有一些指令用来在 FPR 和 GPR 之间移动数据。

12.1.2 数据类型

数据类型包括整型 8 位字节、16 位半字、32 位字、浮点 32 位单精度、64 位双精度。半字类型被加入最小指令集是因为它在一些语言(如 C 语言)中出现,在一些程序(如操作系统程序)中,半字也很流行,因为这些程序很重视数据结构的大小。单精度浮点操作数也因同样的原因而被加入在指令集中。(注意在本书前面提到的原则:在设计指令集之前必须测试更多的程序)。

DLX 的操作是面向 32 位整型数和 32 位或 64 位浮点数的。字节和半字被 load 到寄存器上的时候,用零或者符号位填充 32 位寄存器的剩余部分。一旦被 load,就可以用 32 位整数指令来操作它们。

12.1.3 数据转移寻址模式

数据寻址模式只有均为 16 位的立即寻址和变址寻址。寄存器寻址模式可以通过把 16 位变址域置 0 来实现,16 位绝对地址寻址通过把寄存器 0 作为基址寄存器来实现。于是我们得到了四种有效的寻址模式,尽管体系结构只支持两种。

DLX 内存是用 32 位地址字节寻址的。作为 load/store 结构,所有的内存访问都是通过内存与 GPRs 或 FPRs 之间的 load 或 store 来完成的。除了支持上述的数据类型,内存访问还包括 GPRs 到一个字节,一个半字或一个字。FPRs 可以用单精度或双精度字 load 和 store(双精度数需要一对 FPRs)。所有的内存访问都必须是对准的。

12.1.4 指令格式

因为 DLX 只有两种寻址模式,所以它们可以编码到操作码中。为了使机器更容易进行流水线操作和解码,所有指令都是 32 位的,其起始 6 位是主操作码。图 12.1 展示了指令配置。这些指令格式很简单,同时还为变址寻址、立即数或 PC 相关跳转地址提供了 16 位域。

12.1.5 操作

DLX 支持本文前面所述的一些简单操作,还有一些其它操作。指令大致可分为四种:load 和 store,ALU 操作,分支和跳转,浮点操作。

任何一种通用寄存器和浮点寄存器都可以被 load 和 store,但是 load R0 没有意义。单精度浮点数占用一个浮点寄存器,而双精度浮点数占据一对。单精度和双精度之间的转换必须显式地进行。浮点格式是 IEEE754。图 12.2 给出了 load 和 store 指令的例子。完整的指令序列见图 12.5。为了理解这些图,我们需要介绍一下我们的描述语言。

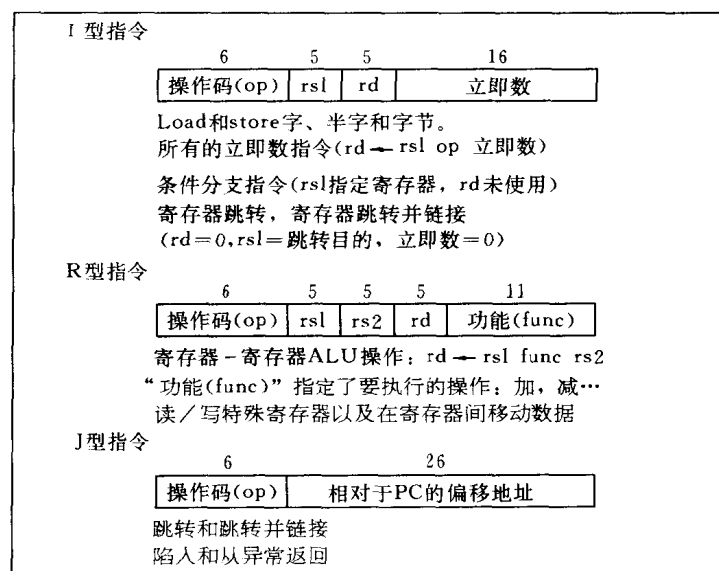


图 12.1 DLX 指令结构(所有指令都具有以上三种类型之一)

指令示例	指令名	含 义
LW R1,30(R2)	Load word	$\text{Regs}[\text{R1}] \leftarrow_{32} \text{Mem}[30 + \text{Regs}[\text{R2}]]$
LW R1,1000(R0)	Load word	$\text{Regs}[\text{R1}] \leftarrow_{32} \text{Mem}[1000 + 0]$
LB R1,40(R3)	Load byte	$\text{Regs}[\text{R1}] \leftarrow_{32} (\text{Mem}[40 + \text{Regs}[\text{R3}]]_0)^{24} \# \# \text{Mem}[40 + \text{Regs}[\text{R3}]]$
LBU R1,40(R3)	Load byte unsigned	$\text{Regs}[\text{R1}] \leftarrow_{32} 0^{24} \# \# \text{Mem}[40 + \text{Regs}[\text{R3}]]$
LH R1,40(R3)	Load half word	$\text{Regs}[\text{R1}] \leftarrow_{32} (\text{Mem}[40 + \text{Regs}[\text{R3}]]_0)^{16} \# \# \text{Mem}[40 + \text{Regs}[\text{R3}]] \# \# \text{Mem}[41 + \text{Regs}[\text{R3}]]$
LF F0,50(R3)	Load float	$\text{Regs}[\text{F0}] \leftarrow_{32} \text{Mem}[50 + \text{Regs}[\text{R3}]]$
LD F0,50(R3)	Load double	$\text{Regs}[\text{F0}] \# \# \text{Regs}[\text{F1}] \leftarrow_{64} \text{Mem}[50 + \text{Regs}[\text{R2}]]$
SW 500(R4), (R3)	Store word	$\text{Mem}[500 + \text{Regs}[\text{R4}]] \leftarrow_{32} \text{Regs}[\text{R3}]$
SF 40(R3),F0	Store float	$\text{Mem}[40 + \text{Regs}[\text{R3}]] \leftarrow_{32} \text{Regs}[\text{F0}]$
SD 40(R3),F0	Store double	$\text{Mem}[40 + \text{Regs}[\text{R3}]] \leftarrow_{32} \text{Regs}[\text{F0}]$ $\text{Mem}[44 + \text{Regs}[\text{R3}]] \leftarrow_{32} \text{Regs}[\text{F1}]$
SH 502(R2),R3	Store half	$\text{Mem}[502 + \text{Regs}[\text{R2}]] \leftarrow_{16} \text{Regs}[\text{R3}]_{16..31}$
SB 41(R3),R2	Store byte	$\text{Mem}[41 + \text{Regs}[\text{R3}]] \leftarrow_8 \text{Regs}[\text{R2}]_{24..31}$

图 12.2 DLX 中的 load 和 store 指令(所有指令都适用单一寻址模式, 存储器的地址必须是对准的。
所有的数据类型都拥有 load 和 store 指令

(1) 假定 R8 和 R10 是 32 位寄存器: $\text{Regs}[\text{R10}]_{16..31} \leftarrow_{16} (\text{Mem}[\text{Regs}[\text{R8}]]_0)^8 \# \# \text{Mem}[\text{Regs}[\text{R8}]]$ 表示 R8 内容再寻址后的字节被扩展为 16 位后存入 R10 的低半字

(R10 的高半字不变)。

(2) 当被转移数据的长度可能不确切时,附加一个脚注到符号 \leftarrow 上。 \leftarrow_n 表示传送 n 位。

(3) 下标用于标志从域中选出某个特定的位。位从最高位 0 起始位标注。下标可以是一个十进制数字(如 $\text{Regs}[\text{R4}]_0$ 表示 R4 的标志位),也可以是一个范围(如 $\text{Reg}[\text{R3}]_{24..31}$ 表示 R3 的最低字节)。

(4) 变量 Mem 用来代表主存,由按字节编址,可以传送任意长度的数据。

(5) 上标用来表示对域进行复制(如 0^{24} 表示一个长度为 24 的全零域)。

(6) $\# \#$ 被用来连接两个域,它可以出现在数据转移的任一方。

所有 ALU 指令都是寄存器到寄存器的指令,包括简单的算术操作和逻辑的操作:加,减,与,或,异或和移位。所有指令都提供了立即寻址方式(包括一个符号位扩展的 16 位立即数)。LHI(高位立即 load)操作 load 一个寄存器的高位部分,并同时置低位置 0。这使得一个 32 位的常数可以用两条指令来建立,或通过一条额外的指令中指定 32 位常数地址进行数据移动来得到。

如上所述,R0 被用来合成通用操作。load 一个常量的操作可以通过一个源操作数是 R0 的立即寻址来完成。寄存器到寄存器移动也可以通过其中一个源操作数是 R0 的加法来完成。(我们有时用 LI 表示立即方式的 load 来指代前者,用 MOV 来指代后者。)

比较指令对两个寄存器的值进行比较($=, \neq, <, >, \leq, \geq$)。如果条件为真,则指令置目的寄存器为 1(表示真),否则置 0。因为那些操作“置”一个寄存器,它们被称为置等、置不等、置小于等等。这些比较指令也提供了立即寻址的方式。图 12.3 给出了算术和逻辑指令的例子。

指令示例	指令名	含 义
ADD R1,R2,R3	加(Add)	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + \text{Regs}[\text{R3}]$
ADDI R1,R2,#3	加立即数 (Add immediate)	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + 3$
LHI R1,#42	Load 立即数到高半字 Load high immediate	$\text{Regs}[\text{R1}] \leftarrow 42 \# \# 0^{16}$
SLLI R1,R2,#5	左移逻辑立即数 (Shift left logical immediate)	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] \ll 5$
SLT R1,R2,R3	置小于 (Set less than)	$\text{if}(\text{Regs}[\text{R2}] < \text{Regs}[\text{R3}]) \text{Regs}[\text{R1}] \leftarrow 1$ $\text{else } \text{Regs}[\text{R1}] \leftarrow 0$

图 12.3 DLX 中的算术和逻辑指令例子,包括立即寻址方式和非立即寻址方式

跳转和分支指令实现控制功能。四种跳转指令由指令的目的地址的两种方式 and 是否进行链接来区分。两种跳转用 26 位有符号数附加到程序计数器(紧跟跳转的下一条指令的地址)来决定目的地址,而另外两种跳转用包含目的地址的寄存器来决定。跳转有两种方式:简单跳转和跳转并链接(用于程序调用)。后者把返回地址放入寄存器 R31。

所有的分支都是条件分支指令。分支条件由指令来决定,指令通过检查其指定的源寄存器是否为零来决定分支与否,该寄存器可能是一个数据或是比较的结果。分支的目的地

址用附加到程序计数器上(PC)的 16 位有符号偏移地址来指定。图 12.4 给出了典型的分支和跳转指令。后面我们介绍浮点条件分支指令,该指令通过测试浮点寄存器状态寄存器来决定是否分支。

指令示例	指令名	含 义
J name	跳转 (Jump)	$PC \leftarrow name;$ $((PC+4)-2^{25}) \leq name < ((PC+4)+2^{25})$
JAL name	跳转并链接 (Jump and link)	$R31 \leftarrow PC+4; PC \leftarrow name;$ $((PC+4)-2^{25}) \leq name < ((PC+4)+2^{25})$
JALR R2	寄存器跳转并链接 (Jump and link register)	$Regs[R3] \leftarrow PC+4; PC \leftarrow Regs[R2]$
JR R3	寄存器跳转 (Jump register)	$PC \leftarrow Regs[R3]$
BEQZ R4, name	等于 0 时分支 (Branch equal zero)	$if (Regs[R4] = 0) PC \leftarrow name;$ $((PC+4)-2^{25}) \leq name < ((PC+4)+2^{25})$
BNEZ R4, name	不等于 0 时分支 (Branch not equal zero)	$if (Regs[R4] \neq 0) PC \leftarrow name;$ $((PC+4)-2^{25}) \leq name < ((PC+4)+2^{25})$

图 12.4 典型的 DLX 控制流指令(所有的控制指令,除了跳转到寄存器指定地址的指令外,都是相对于 PC 的。如果用 R0 作为寄存器操作数,那么 BEQZ 总会产生分支,但是编译器通常会使用具有较长偏移地址的跳转指令来表示这种“无条件分支”)

浮点指令对浮点寄存器进行操作并指示要执行的操作是单精度还是双精度。MOVE 和 MOVD 分别复制单精度(MOVF)和双精度(MOVD)到另一个同类型寄存器。MOVFP2I 和 MOVI2FP 分别在一个单精度寄存器和一个整型寄存器之间移动数据。而移动一个双精度数据到两个整型寄存器需要两条指令。在 32 位浮点寄存器上的整型乘和除用来提供从整型到浮点数的转换,反之亦然。

浮点操作包括加、减、乘、除。后缀为 D 表示双精度,后缀为 F 表示为单精度(如 ADD,ADDF,SUBD,SUBF,MULTD,MULTF,DIVD,DIVF)。浮点比较指令会设置一个特殊的浮点状态寄存器。浮点分支指令 BFPT 和 BFPF 通过测试浮点状态寄存器来决定是否进行分支。

DLX 的一个不太寻常的特点是它用浮点单元作整型乘除。我们在本书中看到,浮点操作控制要比整型加减复杂得多。因为浮点单元早就可以处理浮点乘除,所以来处理相对较慢的整型乘除并不困难。因此,DLX 要求把进行乘除的操作数放入浮点寄存器。

图 12.5 包含了所有 DLX 操作和含义的列表。为了说明哪些操作更加常用,图 12.6 说明了 5 个 SPECint92 程序的指令和指令类的调用频度。图 12.7 展示了 SPECFP92 的数据。图 12.8 和图 12.9 展示了平均使用频率超过 1% 的指令的使用频度图,以给读者一个更直观的印象。

指令类型/操作码	指令含义
数据传送	在寄存器或内存之间移动数据,或是在整数寄存器与浮点寄存器或特殊寄存器之间移动数据唯一的寻址模式是 16 位偏移地址+通用寄存器(GPR)内容
LB,LBU,SB	Load 字节(Load Byte),Load 无符号字节(Load Byte Unsigned),Store 字节(Store Byte)
LH,LHU,SH	Load 半字(load Half Word),Load 无符号半字(Load Half Word Unsigned),Store 半字(Store Half Word)
LW,SW	Load 字(Load Word),Store 字(Store Word)(整数寄存器操作)
LF,LD, SF,SD	Load 单精度浮点数(Load SP Float),Load 双精度浮点数(Load DP Float),Store 单精度浮点数(Store SP Float),Store 双精度浮点数(Store DP Float)
MOVI2S,MOVS2I	在通用寄存器(GPR)与特殊寄存器之间移动数据
MOVFP,MOVFP2I,MOVFP2I,MOVFP2I	复制一个 FP 寄存器或一对 DP 寄存器的内容到另一个寄存器或寄存器对在整数寄存器和 FP 寄存器间移动 32 位数据
算术/逻辑	在 GPR 中的整数或逻辑数据的操作;有符号算数运算在溢出时发生陷入
ADD,ADDI,ADDU,ADDUI	加(Add),加立即数(Add Immediate)(所有立即数都是 16 位),有符号加和无符号加。
SUB,SUBI,SUBU,SUBUI	减,减立即数,有符号减和无符号减。
MULT,MULTU,DIV,DIVU	乘,除,包括有符号运算和无符号运算;操作数必须是 FP 寄存器;所有操作使用并得到 32 位的值
AND,ANDI	与(And),与立即数(And Immediate)
OR,ORI,XOR,XORI	异或(Exclusive Or),异或立即数(Exclusive Or Immediate)
LHI	Load 立即数到高半字(Load High Immediate)
SLL,SRL,SRA,SLLI, SRLI,SRAI	左移逻辑值(Shift Left Logical),右移逻辑值(Shift Right Logical),右移算术值(Shift Right Arithmetic);移位立即数(S-I)
S_-,S-I	设置条件;“_”可以是 LT(小于),GT(大于),LE(小等于),GE(大等于),EQ(等于),NEQ(不等于)
控制	条件分支或跳转;相对于 PC 或是由寄存器指明跳转地址
BEQZ,BNEZ	GRP 等于 0/不等于 0 时跳转;地址是相对于 PC+4 的 16 位偏移地址
BFPT,BFPF	测试 FP 状态寄存器中的比较位并分支;地址为相对于 PC+4 的 16 位偏移地址
J,JR	跳转(Jump);目的地址为相对于 PC+4 的 26 位偏移地址(J)或由寄存器指定(JR)
JAL,JALR	跳转并链接(Jump And Link);把 PC+4 保存在 R31 中,目的地址为相对于 PC 的偏移地址(JAL)或由寄存器指定(JALR)
TRAP	陷入
RPE	从异常中返回(Return From An Exception)恢复到用户态
浮点	浮点操作
ADDD,ADDF	加 DP,SP 浮点数
SUBD,SUBF	减 DP,SP 浮点数
MULTD,MULTF	乘 DP,SP 浮点数
DIVD,DIVF	除 DP,SP 浮点数
CVTF2D,CVTF2I, CVTD2F,CTD2I,	数据类型转换指令: Cvt _x 2 _y 从类型 X 转到类型 Y,其中 X 和 Y 是 I(整数 Integer),D(双精度浮点数 Double Precision)或 F(单精度浮点数 Single Precision)。
CVTI2F,CVTI2D	两个操作数都是浮点寄存器(FPR)
D_-,F	DP 和 SP 比较指令;“_”可以是 LT(小于),GT(大于),LE(小等于),GE(大等于),EQ(等于),NEQ(不等于)并在 FP 状态寄存器中置位。

图 12.5 DLX 指令集

指令示例	compress	eqntott	espresso	gcc(cc1)	li	占 SPECint92 的平均百分比
load	30.6%	20.9%	22.8%	31.3%	26%	19.8%
store	5.6%	0.6%	5.1%	14.3%	16.7%	9%
加	14.4%	8.5%	23.8%	14.6%	11.1%	14%
减	1.8%	0.3%		0.5%		0%
乘				0.1%		0%
除						0%
比较	15.4%	26.5%	8.3%	12.4%	5.4%	13%
load 立即数	8.1%	1.5%	1.3%	6.8%	2.4%	3%
条件分支	17.4%	24.0%	15.0%	11.5%	14.6%	16%
无条件分支	1.5%	0.9%	0.5%	1.3%	1.8%	1%
调用	0.1%	0.5%	0.4%	1.1%	3.1%	1%
返回、间接跳转	0.1%	0.5%	0.5%	1.5%	3.5%	1%
移位	6.5%	0.3%	7.0%	6.2%	0.7%	4%
与	2.1%	0.1%	9.4%	1.6%	2.1%	3%
或	6.0%	5.5%	4.8%	4.2%	6.2%	5%
其他(异或,非)	1.0%		2.0%	0.5%	0.1%	1%
浮点 load						0%
浮点 store						0%
浮点加						0%
浮点减						0%
浮点乘						0%
浮点除						0%
浮点比较						0%
浮点寄存器						0%
其他浮点指令						0%

图 12.6 5 个 SPECint 92 程序中的 DLX 各类指令所占比例(整数寄存器之间的移动指令算在 add 指令中,空表项的值为 0.0%)

12.1.6 效率

用简单指令格式、简单寻址模式的简单操作的指令结构看起来可能会比较慢,部分原因是它必须执行比复杂指令结构更多的指令。但是性能方程式提醒我们,执行时间是一个函数而不是简单的指令条数。

$$\text{CPU 时间} = \text{指令条数} \times \text{CPI} \times \text{时钟周期}$$

为了看指令条数的减少的代价是否是 CPI 或时钟周期的增加,我们比较一下 DLX 和一个复杂指令集结构。VAX 是一个很好的复杂指令集结构的例子。在 20 世纪 70 年代

指令示例	docuc	ear	hydro2d	mdljdp2	su2cor	占 SPECfp92 的平均百分比
load	1.4%	0.2%	0.1%	1.1%	3.6%	1%
store	1.3%	0.1%		0.1%	1.3%	1%
加	13.6%	13.6%	10.9%	4.7%	9.7%	11%
减	0.3%		0.2%		0.7%	0%
乘						0%
除						0%
比较	3.2%	3.1%	1.2%	0.3%	1.3%	2%
load 立即数	2.2%		0.2%	2.2%	0.9%	1%
条件分支	8.0%	10.1%	11.7%	9.3%	2.6%	8%
无条件分支	0.9%	0.4%		0.4%	0.1%	0%
调用	0.5%	1.9%			0.3%	1%
返回、间接跳转	0.6%	1.9%			0.3%	0%
移位	2.0%	0.2%	2.4%	1.3%	2.3%	2%
与	0.4%	0.1%			0.3%	0%
或		0.2%	0.1%	0.1%	0.1%	0%
其他(异或,非)						0%
浮点 load	23.2%	19.8%	24.1%	25.9%	21.6%	23%
浮点 store	5.7%	11.4%	9.9%	10.0%	9.8%	9%
浮点加	8.8%	7.3%	3.6%	8.5%	12.4%	8%
浮点减	3.8%	3.2%	7.9%	10.4%	5.9%	6%
浮点乘	12.0%	9.6%	9.4%	13.9%	21.6%	13%
浮点除	2.3%		1.6%	0.9%	0.7%	1%
浮点比较	4.2%	6.4%	10.4%	9.3%	0.8%	6%
浮点寄存器	2.1%	1.8%	5.2%	0.9%	1.9%	2%
其他浮点指令	2.4%	8.4%	0.2%	0.2%	1.2%	2%

图 12.7 5 个 SPECfp92 程序中的 DLX 各类指令所占比例(整数寄存器之间的移动指令算在 add 指令中,空的表项的值为 0.0%)

中期,当 VAX 被设计出来时,当时流行的观点是建立与程序语言相近的指令集来简化编译器。例如,因为编程语言有循环,指令集就应该有循环指令,而不是只有简单跳转指令;还需要调用指令来节约寄存器而不是简单的跳转和链接;还需要 case 指令,而不是间接跳转,等等。同样的道理,VAX 提供了一套庞大的寻址模式并确保所有的指令都可以使用这些寻址模式。另外一个盛行的设计思想是使代码长度最小化。让我们回想一下 DRAM 的容量每三年扩大四倍,因此 70 年代的 DRAM 芯片的容量比现在的 1%还少,所以代码空间也很关键。代码空间在固定长度指令集如 DLX 中被削弱了重要性。例如,DLX 地址空间一直占用 16 位,尽管有时地址很小。相反,VAX 允许指令长度可变,所以空间浪费

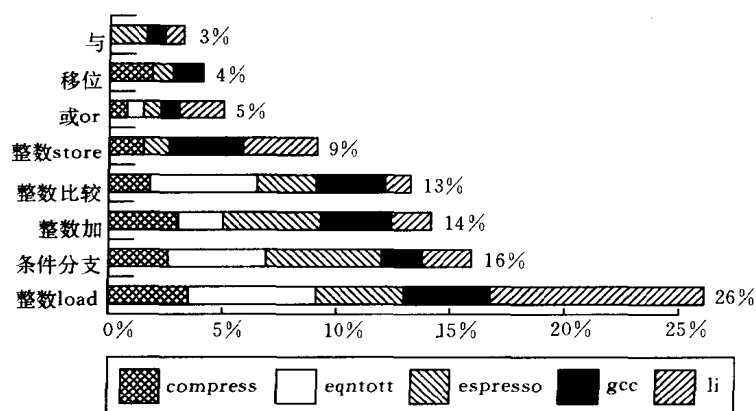


图 12.8 图 12.6 中 5 个 SPECint92 程序中指令执行状况的图形表示

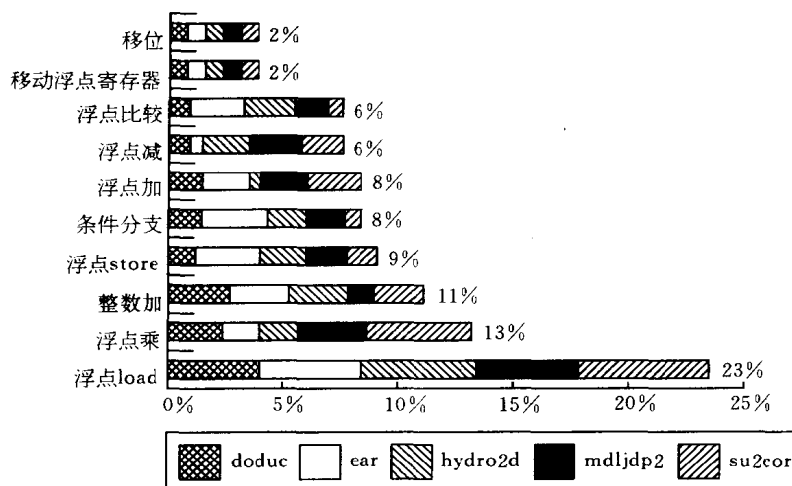


图 12.9 图 12.7 中 5 个 SPECfp92 程序中指令执行状况的图形表示

较少。

VAX 设计者后来对 VAX 和类 DLX 处理器进行了一次定量比较。他们选的是 VAX 8700 和 MIPS M2000。不同的设计目的导致了它们的不同的体系结构。VAX 的设计目的是使用简单的编译器和减小执行代码的长度，因此它有强大的寻址模式和指令，有效指令编码，较少的寄存器。MIPS 的目的是通过流水线获得高性能的计算能力，易于用硬件实现，与高度优化的编译器协同工作。这些目的导致了精简指令、简单寻址模式、固定长度指令格式和大量寄存器。

图 12.10 展示了指令执行数比率、CPI 比率、一个时钟周期内的性能比率。因为组织结构相似，时钟周期被认为一致。MIPS 执行的指令条数大约是 VAX 的两倍，而 VAX 的 CPI 比 MIPS 的长 6 倍，因此 MIPS M2000 实际上有三倍于 VAX 8700 的性能。而且 MIPS 的 CPU 对硬件的需求比 VAX 的 CPU 要少得多。这种性能价格比的差距正是许多过去制作 VAX 的公司渐渐放弃了 VAX 而改做类似于 DLX 的机器的原因。

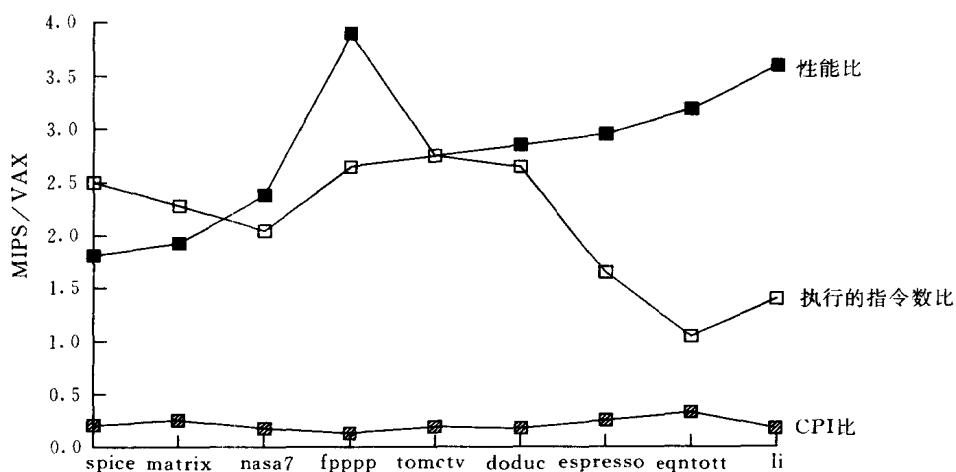


图 12.10 MIPS M2000 和 VAX 8700 在运行 Spec 89 程序时的性能和执行指令数之比(MIPS 执行的指令条数大约是 VAX 的两倍,而 VAX 的 CPI 比 MIPS 的长 6 倍,因此 MIPS M2000 实际上有三倍于 VAX 8700 的性能。(根据 Bhandakar 和 Clark 的数据[1991]))

12.2 DLX 流水线结构

12.2.1 一个简单实现

为了理解 DLX 怎样进行流水线操作,首先要弄清楚在不进行流水时它是如何实现的。这一节给出了 DLX 实现的一个简单的例子,在这个例子中,所有的指令至多占用五个时钟周期。该实现方案扩展到流水线之后,CPI 将大大地降低。尽管该方案并不是最经济 and 最高效的非流水线实现方案,但是其设计可以自然地扩展到流水线版本。实现指令集的时候需要一些并不属于这个体系结构的临时寄存器,我们引入这些寄存器以简化流水线。

每一条 DLX 指令的实现至多需要五个时钟周期。这五个时钟周期如下:

1. 取指周期(IF)

$$IR \leftarrow \text{Mem}[PC]$$

$$NPC \leftarrow PC + 4$$

操作:根据 PC 指示的地址从存储器中取指并放入到指令寄存器(IR),同时 PC 加 4 以获取下一条指令的地址。IR 中 store 下一个时钟周期需要的指令;同样,NPC 中 store 下一个 PC。

2. 分析指令/取寄存器周期(ID)

$$A \leftarrow \text{Regs}[IR_{6..10}] ;$$

$$B \leftarrow \text{Regs}[IR_{11..15}] ;$$

$$\text{Imm} \leftarrow ((IR_{16})^{16} \# \# IR_{16..31})$$

操作:分析指令并访问寄存器堆以读取寄存器。通用寄存器的输出送入两个临时的寄存器(A 和 B)中以供后面的时钟周期使用。IR 的低 16 位也进行符号扩展并存入到临时寄存器 Imm 中以供下一个时钟周期使用。

因为 DLX 指令有着固定的格式(图 12.1),所以读寄存器和分析指令是可以并行执行的。这项技术被称为固定域解码(fixed-field decoding)。大家会注意到我们可能会读取了寄存器但并未使用读取的值,这虽然没有什么好处但也没有什么妨碍。由于指令的立即数部分在 DLX 格式中都有相同的位置,因此在下一个周期需要时,带符号扩展的立即数也在当前时钟周期进行计算。

3. 执行/有效地址周期(EX)

ALU 对上一个时钟周期预备好的操作数进行操作,根据 DLX 指令的类型执行下面四个功能中的一个:

- 存储器引用

$$\text{ALUOutput} \leftarrow A + \text{Imm};$$

操作: ALU 通过加法运算形成有效地址并将结果放入到寄存器 ALUOutput 中。

- 寄存器-寄存器 ALU 指令:

$$\text{ALUOutput} \leftarrow A \text{ op } B$$

操作: ALU 根据操作码对寄存器 A 和寄存器 B 中的数值进行操作。结果被放入到临时寄存器 ALUOutput 中。

- 寄存器-立即数 ALU 指令:

$$\text{ALUOutput} \leftarrow A \text{ op } \text{Imm};$$

操作: ALU 根据操作码对寄存器 A 和寄存器 Imm 中存放的值进行操作。结果放入到临时寄存器 ALUOutput 中。

- 分支

$$\text{ALUOutput} \leftarrow \text{NPC} + \text{Imm}$$

$$\text{cond} \leftarrow (A \text{ op } 0)$$

操作: ALU 将 NPC 同 Imm 中的带符号立即数相加以计算分支的目标地址。在前一个时钟周期读取的寄存器 A 的值用于决定是否进行分支操作。比较操作 op 是分支操作码所决定的关系操作符。

DLX 的 load/store 结构意味着有效地址周期和指令执行周期可以结合成一个时钟周期,这是因为没有指令需要同时计算数据地址、指令目标地址和对数据进行操作。除了上面的指令外,还有一些不同形式的跳转指令,它们与分支指令类似。

4. 存储器访问/分支完成周期(MEM)

在这个周期能进行操作的 DLX 指令仅包括: load、store 和分支。

- 存储器引用

$$\text{LMD} \leftarrow \text{Mem}[\text{ALUOutput}] \text{ 或}$$

$$\text{Mem}[\text{ALUOutput}] \leftarrow B$$

操作: 在需要时访问存储器。如果是装入指令,将从存储器返回数据并放入 LMD (load memory data)寄存器;如果是 store 指令则将寄存器 B 中的数据写入存储器。这两

种情况下使用的地址在上一个周期里计算得到并放在寄存器 ALUOutput 中。

- 分支：

$\text{if (cond) PC} \leftarrow \text{ALUOutput else pc} \leftarrow \text{NPC}$

操作：如果进行分支操作，PC 将为寄存器 ALUOutput 中 store 的分支的目标地址所替代；否则被寄存器 NPC 中新的经过自增的 PC 替代。

5. 回写周期(WB)

- 寄存器-寄存器 ALU 指令

$\text{Regs}[\text{IR}_{16..20}] \leftarrow \text{ALUOutput}$

- 寄存器-立即数 ALU 指令：

$\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{ALUOutput} ;$

- 装入指令

$\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{LMD} ;$

操作：将结果写入寄存器堆，它可能来自于存储器系统(存放在 LMD 中)或者来自于 ALU(存放在 ALUOutput 中)；寄存器目标区域在两个位置中选择，它依赖于操作码。

图 12.11 给出了一条指令如何沿着数据通路进行流动。在每一个时钟周期结束时，所有的在该时钟周期计算得到并在后面的时钟周期需要使用(不论是本条指令或者是下一条指令)的数据将被写入到 store 设备，它可能是存储器、通用寄存器、PC 或者临时寄存器(即 LMD, Imm, A, B, IR, NPC, ALUOutput 和 Cond)。临时寄存器为一条指令在不同的时钟周期之间维持数值，而其他的寄存器则是状态的可见部分，它在相继的指令之间维持数值。

在这个实现中，分支指令需要四个时钟周期，而其它指令需要五个时钟周期。假定分支的频率为 12% 的话，那么总体的 CPI 将为 4.88。这个实现在获取高性能或者在相同性能的条件下的使用最少硬件方面都不是最优的。在不改变时钟频率的条件下，CPI 可以通过在 MEM 周期完成 ALU 指令来降低，因为这些指令在该周期是闲置的。按照第二章所测，假定 ALU 指令在混合指令中占 44% 的比例，使用这种方法将会是 CPI 降为 4.44，减小为原来的 1/1.11。除了这种简单的方法外，其他减少 CPI 的方法都会增加时钟周期的长度，因为这些方法需要每一个时钟周期做更多的动作。当然，以增加时钟周期的长度为代价来换取 CPI 的降低也许还是合算的，但是这需要仔细地进行分析，特别是当操作在各个时钟周期的初始分布已经相当平衡时，不会有太大的改善。

虽然今天所有的机器都使用流水线技术，但是这个多周期的例子也大致反映了在早些时候大多数机器是怎样进行实现的。一个简单的有限状态机可以使用上面的五个周期的结构进行控制。对于一个更为复杂的机器，可以使用微码控制。在这两种情况下，上面所述的指令序列都决定了控制的结构。

除了降低 CPI 外，在这个多周期的实现中还有一些硬件冗余可以进行消除。例如，这个结构中有两个 ALU：一个进行 PC 的自加运算，另一个进行有效地址和 ALU 计算。由于它们并不需要在同一个时钟周期使用，我们可以通过增加一个转换开关将它们合并，使之共享同一个 ALU。类似的，指令和数据可以存放在相同的存储器中，因为数据和指令存取在不同的时钟周期发生。

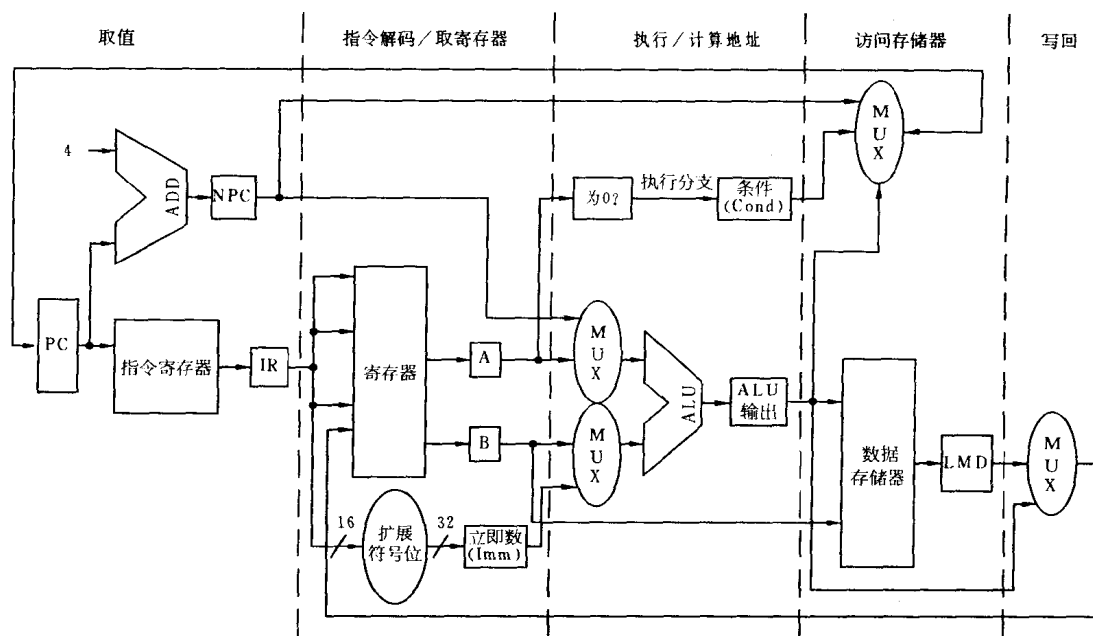


图 12.11 DLX 的数据通路的实现,它允许在 4 或 5 个时钟周期内完成一条指令。虽然这些功能部件出现在它们被读出的周期,但是 PC 在存储器访问周期(同样在取指周期)被写入,而寄存器在回写周期被写入。在这两种情况下,在后面节拍的写入可以由回送数值给 PC 或者寄存器的开关输出(在存储器访问和回写周期)来指明。这些回流的信号大大地增加了流水线的复杂度,我们将在下几节中进行仔细讨论。

我们宁愿保留图 12.11 的设计而不进行优化,因为它为我们流水线化的实现提供了一个良好的基础。

除了这一节讨论的多周期设计外,我们也可以让每一条指令占用一个长的时钟周期。在这种情况下,所有的临时寄存器都可以抛弃不用,因为在一条指令的内部时钟周期间没有进行通信。每一条指令都在一个长的时钟周期内执行,并在该时钟结束时将结果写入存储器、寄存器或者 PC。对于这个机器来说,CPI 将始终是 1。但是,其时钟周期大致等于多周期机的 5 倍,因为每一条指令都要通过所有的功能部件。设计者不会使用这种单周期的实现方法,因为有两方面的原因:首先,对于大多数机器来说,不同的指令需要的时钟周期的时间不同,单周期的实现方法效率很低;其次,单周期的实现方法需要重复的功能部件,而在多周期的实现中功能部件可以共享。不过,这个单周期的数据路径可以用来解释流水线技术是如何改善机器的时钟周期,而不是 CPI 的。

12.2.2 基本流水线

我们只需在每一个时钟周期启动一条新的指令便可以使图 12.11 所示的数据通路流起来(由此可以看到我们为什么选择这个实际方案)。前一节中的每一个时钟周期便成了一个流水节拍:流水线的的一个周期。这样便有图 12.12 所示的指令执行模式,它使用

流水结构的典型画法。其中，每一条指令经过五个时钟周期执行完成，在每一个时钟周期内，硬件将初始化一条新的指令并执行五个不同指令的某个部分。

	时 钟 数								
指令数	1	2	3	4	5	6	7	8	9
指令 i	IF	ID	EX	MEM	WB				
指令 $i+1$		IF	ID	EX	MEM	WB			
指令 $i+2$			IF	ID	EX	MEM	WB		
指令 $i+3$				IF	ID	EX	MEM	WB	
指令 $i+4$					IF	ID	EX	MEM	WB

图 12.12 一个简单的 DLX 流水线。在每一个时钟周期都有一条新的指令取进来并开始长达五个时钟周期的执行。若在每一个时钟周期都启动一条新的指令，那么性能将是不进行流水的机器的五倍。流水线每一个节拍的名称为：IF=取指令，ID=分析指令，EX=执行指令，MEM=存储器访问，WB=写回。

也许你难以理解流水线竟然会如此简单，你的直觉是对的，实际上它并非如此。在这一节和后面的几节里，我们将讨论一些因为流水而引入的问题，从中你可以对 DLX 流水线有一个真正的认识。

首先需要确定机器在每一个时钟周期都进行什么样的动作，并保证在同一个时钟周期没有两条指令使用相同的数据通路资源。例如，一个 ALU 不能同时用于计算有效地址和减法运算。因此，我们必须保证流水线中指令的重叠不会导致这样的冲突。幸运的是，由于这个 DLX 指令集比较简单，因此资源评估比较容易。图 12.13 给出了简化的 DLX 数据通路。你可以看到，主要的功能部件都在不同的时钟周期使用。因此，多条指令的重叠执行相对来说几乎没有引入冲突。这从下面三点可以看出。

首先，上一节给出的基本的数据通路已经使用了分开的指令和数据存储器，其典型的实现方式是使用分开的指令和数据 Cache。使用独立的 Cache 避免了对单一存储器进行取指和访问数据操作之间的冲突。应该注意的是，如果我们的流水线机的时钟周期和未流水的机器相同时，存储系统的带宽需要是原先的五倍，这便是取得高性能的代价。

其次，寄存器堆在两个流水节拍中使用：ID 节拍读，WB 节拍写。这些使用是截然不同的，因此我们简单地在两个地方画出了寄存器堆。这确实意味着在一个时钟周期需要执行两次读和一次写。如果对同一个寄存器进行读和写将会怎样呢？在这里，我们先将这个问题搁置起来，在下一节进行讨论。

最后，图 12.13 并没有涉及到 PC。为了在每一个时钟周期都启动一条新的指令，需要对 PC 进行自加运算并存回，这项工作必须在 IF 节拍完成，为下一条指令做准备。如果考虑到分支的影响，问题就产生了，因为它也要改变 PC，但是在 MEM 节拍进行。在我们的多周期的非流水线结构中，这不成为一个问题，因为只在 MEM 节拍写一次 PC。在这里我们将只在 IF 节拍对 PC 进行写操作，写入自增后的 PC 或者前面分支的目标地址。这种变化引入 如何处理分支的问题，这一点我们将在后面进行讨论。

由于在每一个时钟每一个流水节拍都是活动的，一个流水节拍的所有操作都必须在

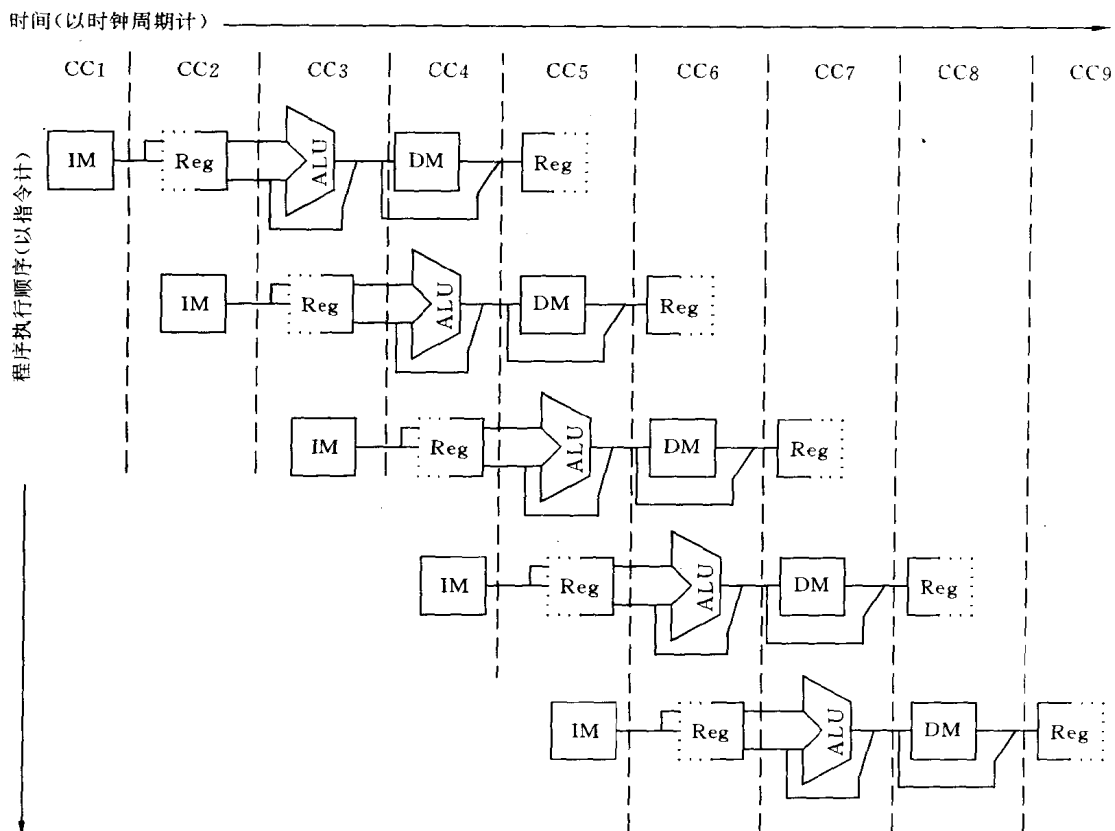


图 12.13 流水线可以看作是随时间进行移位的一系列数据路径(寄存器堆在 EX 节拍要读、在 WB 节拍要写所以出现了两次。包围该节拍的线框中,如果实线在右侧,说明是读操作,如果实线在左侧,说明是写操作;缩写 IM 表示指令存储器,DM 表示数据存储器,CC 表示时钟周期)

一个时钟周期内完成,并且任何操作组合都可以立即出现。另外,数据通路流水要求从一个流水节拍传下一个流水节拍的数据必须放在寄存器中。图 12.14 给出了 DLX 流水相关的各个流水节拍间的寄存器,它们称做流水寄存器。它们上面标有所连接的流水节拍。从图 12.14 可以清楚地看出各个流水节拍之间通过流水寄存器的连接。

所有用于在一条指令的各个时钟周期间维持临时数据的寄存器都归入流水线寄存器这一类中。指令寄存器(IR)的各个域(它们是 IF/ID 寄存器的一部分),当它们用于提供寄存器名时将进行标记。流水线寄存器在两个相邻流水节拍之间既传递数据也传递控制信息。后面流水节拍需要的数据需要放在这样的寄存器中并从一个流水寄存器拷贝到下一个流水寄存器,直到它不再需要。如果我们试图使用早先的非流水数据通路中使用的临时寄存器,数据在未使用完之前就会被覆盖掉。例如对于 load 或者 ALU 操作,写用到的寄存器操作数域由 MEM/WB 流水线寄存器提供而不是 IF/ID 寄存器提供。这是因为我们需要一个 load 或者 ALU 操作来写由这个操作指明的寄存器,而不是当前从 IF 节拍转移到 ID 节拍的指令寄存器域。这个目标寄存器域仅简单地从一个流水寄存器复制到

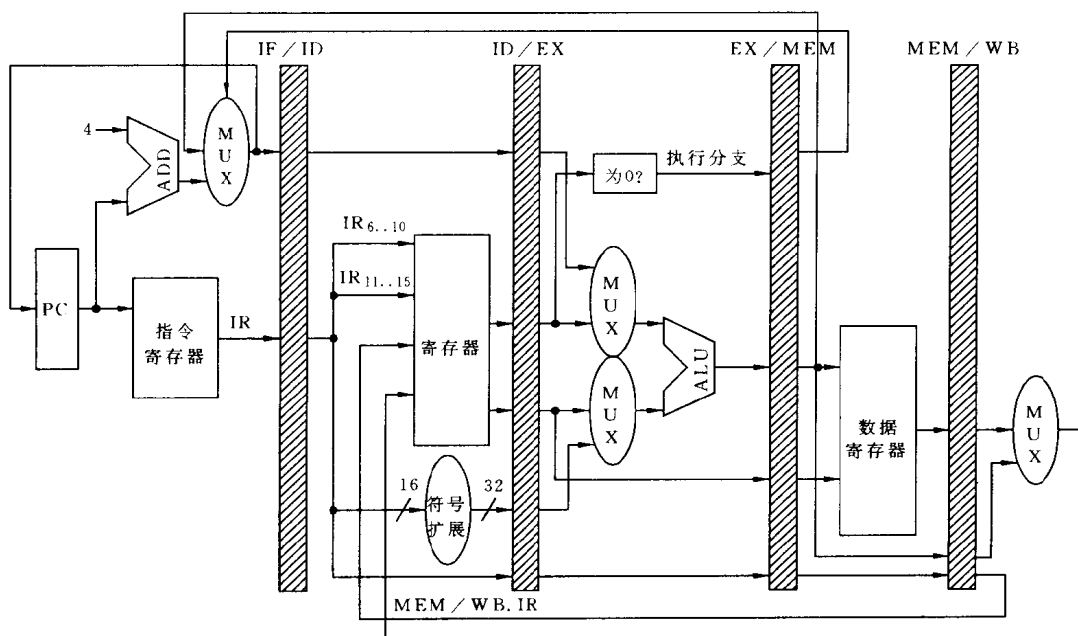


图 12.14 通过各个流水节拍之间增加一系列寄存器进行流水。这些寄存器用于在一个流水节拍和下一个流水节拍之间进行值传递和信息控制。我们也可以将 PC 看成一个流水寄存器,它位于 IF 之前。PC 的多路开关已经经过移动使得 PC 准确地在一个流水节拍 (IF) 写入。如果不移动的话,将会与分支指令发生冲突,因为有两条指令要同时对 PC 写入不同的值。大多数的数据路径随时钟先后从左向右流动。从右向左的流动(载有写回信息和分支的 PC 信息)引入了冲突,需要在后面的章节详细讨论。

下一个流水器,直到在 WB 节拍被使用。

在任意时刻,任何指令都只在一个流水线节拍上是活动的,因此任何指令所作的动作都发生在一对流水线寄存器之间。这样的话,我们便可以根据指令的类型检查在任一流水节拍都要做些什么来看出流水线的动作。图 12.15 给出了这个视图。为了清楚地给出数据从一个流水节拍到下一个流水节拍的流动情况,流水线寄存器域进行了命名。应该注意的是,前两个节拍的動作独立于当前的指令类型,因为指令解码只有到 ID 节拍结束时才进行,所以它们也肯定是独立的。IF 的动作依赖于 EX/MEM 中的指令是否走向分支。如果这样的话,分支的目标地址将既用于取指也用于计算下一个 PC。否则,使用当前的 PC。(我们前面说过,分支的结果导致了流水线的冲突。我们将在后面的章节里讨论。)寄存器源操作数的固定位置编码对于允许在 ID 节拍取寄存器是非常关键的。

为了控制这个简单的流水线,我们只需要决定如何控制图 12.14 中数据路径的四个转换开关。在 ALU 节拍的两个转换开关根据指令类型来设置,它由 ID/EX 寄存器的 IR 与给出。顶上的 ALU 输入转换开关根据指令是否为分支操作来决定,下面的多路开关根据指令是寄存器—寄存器 ALU 操作还是其它类型的操作来决定。IF 节拍的转换开关用

阶段	任 何 指 令		
IF	IF/ID. IR \leftarrow Mem[PC] IF/ID. NPC, PC \leftarrow (if EX/MEM. cond {EX/MEM. NPC} else {PC+4});		
ID	ID/EX. A \leftarrow Regs[IF/ID. IR _{6..10}]; ID/EX. B \leftarrow Regs[IF/ID. IR _{11..15}]; ID/EX. NPC \leftarrow IF/ID. NPC; ID/EX. IR \leftarrow IF/ID. IR; ID/EX. IMM \leftarrow (IR ₁₆) ¹⁶ # IR _{16..31}		
	ALU 指令	Load 或 store 指令	分支指令
EX	EX/MEM. IR \leftarrow ID/EX. IR; EX/MEM. ALUOutput \leftarrow ID/EX. A op ID/EX. B or EX/MEM. ALUOutput \leftarrow ID/EX. A op ID/EX. Imm; EX/MEM. cond \leftarrow 0;	EX/MEM. IR \leftarrow ID/EX. IR EX/MEM. ALUOutput \leftarrow ID/EX. A + ID/EX. Imm; EX/MEM. cond \leftarrow 0; EX/MEM. B \leftarrow ID/EX. B	EX/MEM. ALUOutput \leftarrow ID/EX. NPC + ID. EX. Imm; EX/MEM. cond \leftarrow (ID/EX. A op 0);
MEM	MEM/WB. IR \leftarrow EX/MEM. IR; MEM/WB. ALUOutput \leftarrow EX/MEM. ALUOutput;	MEM/WB. IR \leftarrow EX/MEM. IR; MEM/WB. LMD \leftarrow [EX/MEM. ALUOutput]; or Mem[EX/MEM. ALUOutput] \leftarrow EX/MEM. B;	
WB	Regs[MEM/WB. IR _{16..20}] \leftarrow MEM/WB. ALUOutput; or Regs[MEM/WB. IR _{11..15}] \leftarrow MEM/WB. ALUOutput	Regs[MEM/WB. IR _{11..15}] \leftarrow MEM/WB. LMD	

图 12.15 DLX 流水线的每一个流水线节拍发生的事件。让我们回顾一下这个流水线组织所特有的流水线操作。在 IF 节拍,除了取指和计算新的 PC 外,还将增加后的 PC(NPC)存入流水寄存器以供后面计算分支目标地址时使用。这个结构和图 12.14 的结构相同,PC 在 IF 节拍被一个或者两个数据源更新。在 ID 节拍,进行取寄存器操作,对 IR 的低 16 位进行符号扩展,并传递 IR 和 NPC。在 EX 节拍,进行 ALU 操作或者地址计算工作,传递 IR 和 B 寄存器(如果指令是存入的话)。同时,当指令走向分支时将 cond 的值设置为 1。在 MEM 节拍,进行存储器访问,完成分支,在需要时写 PC,并传递在最后节拍所需的数据。最后,在 WB 节拍,用 ALU 的输出或者装入的值来更新寄存器域。为了简化起见,我们在流水节拍间传递整个 IR,尽管随着指令沿着流水线前进,IR 有用的部分越来越少。

于选择是使用当前的 PC 还是 EX/MEM. NPC 的值(分支的目标地址)作为指令的地址。这个转换开关由 EX/MEM. cond 来控制。第四个转换开关由 WB 节拍的指令是 load 还是 ALU 操作来进行控制。除了这四个转换开关外,还需要一个转换开关,它在图 12.14 中并未画出,仔细看一下 ALU 操作的 WB 节拍便很清楚为什么需要这样一个转换开关。目标寄存器域有两个位置,它依赖于指令类型(是寄存器—寄存器 ALU 还是寄存器—立即数 ALU 指令或装入指令)。因此,我们需要一个转换开关来选择 MEM/WB 寄存器中

IR 的正确部分以指明寄存器的目标域。

对于程序中的数据相关,DLX 处理器采用设置专用通路(forwarding)的方法来解决,详细的信息请参阅本书 5.1.3 节和 WinDLX 联机帮助文件。

12.2.3 扩展 DLX 操作使其能处理多周期操作

我们现在要研究一下如何将我们的 DLX 流水线扩展到可以处理浮点操作。在这一节中,我们将集中讲述基本的策略和可行的方案,另外,我们还附上一个对 DLX 流水线的性能评测。

要求在一个甚至两个时钟周期中完成所有的 DLX 浮点操作,是不现实的。这样做或者意味着大大延长时钟周期,或者意味着在浮点单元中使用大量的庞杂的逻辑,或者两者都是。所以,我们允许浮点流水线操作具有较长的延迟时间。假如我们设想浮点指令具有同整数指令相同的流水线,这就很容易理解了。当然,要有两个重要的改动。首先,为完成操作,EX 循环要视需要重复多次,不同的操作重复的次数有所不同;其次,可能需要多个浮点运算单元。如果要发射的指令会使用一个正在工作中运算单元(如浮点除法单元或寄存器端口),我们称之为结构相关(structural hazard)。和数据相关一样,结构相关也会导致流水线阻塞。

在这一节中,假设我们的 DLX 实现中有四个独立的运算单元:

- (1) 主整数单元,它负责 load、store、整数 ALU 操作和分支;
- (2) 浮点和整数乘法器;
- (3) 浮点加法器,它处理浮点加、减和转换;
- (4) 浮点和整数除法器。

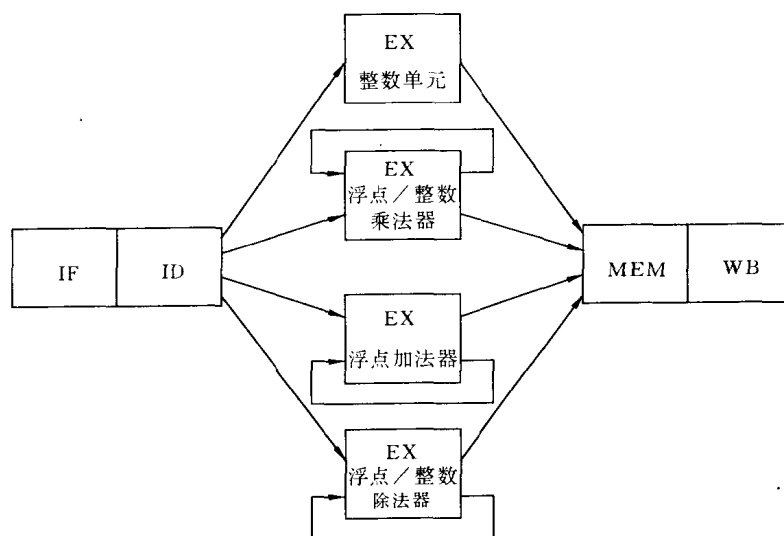


图 12.16 增加了 3 个非流水浮点功能单元的 DLX 流水线(因为每个时钟周期只发射一条指令,所有的指令都经过整数操作的标准流水线。浮点运算仅是在 EX 节拍进行循环。在浮点操作结束了 EX 节拍后,进入 MEM 和 WB 节拍以完成指令)

我们假设这些运算单元的执行(EX)不是流水线化的,图 12.16 显示了它的流水化结构,因为 EX 节拍不是流水化的,所以直到前一条指令离开 EX 节拍之前,其它使用该运算单元的指令都不能执行。另外,如果某条指令不能执行到 EX 节拍,在这条指令之后的整个流水线都会被阻塞。

事实上,中间结果可能并不像那样在 EX 单元循环,而是 EX 的流水节拍有超过一个时钟周期的延迟。

我们可以将图 12.16 所示的浮点流水线的结构推广,以允许某些节拍流水化和多重指令同时操作。为了描述这样的流水线,我们必须定义运算单元的延迟和初始间隔或重复间隔。我们把延迟定义为:介于一条产生结果的指令和一条使用该结果的指令之间的时钟周期数,初始(或重复)间隔定义为执行两个给定类型的操作之间必须经历的周期数。例如,我们将使用如图 12.17 所列的延迟和初始间隔。

功能部件	延时	初始化间隔
整数 ALU	0	1
数据访存指令(整数和浮点 loads)	1	1
浮点加	3	1
浮点乘(包括整数乘)	6	1
浮点除(包括整数除和浮点 sqrt)	24	24

图 12.17 运算单元的延迟和初始间隔

按照延迟的这种定义,整数 ALU 操作的运算结果在下一个时钟周期就可以使用,所以它的延迟为 0;load 的结果在一个周期之后就可以被使用,所以 load 的延迟为 1。因为大多数操作在 EX 节拍开始时就开始使用它们的操作数。所以,延迟通常是指令在产生结果的 EX 节拍之后还要再执行的节拍数。例如,ALU 运算是 0 拍,load 是 1 步。一个主要的例外是 store,它在一个周期后使用被 store 的值,所以其延迟要少一个周期。流水线延迟基本上等于执行流水线的深度减去一个时钟周期,而执行流水线的深度为 EX 节拍到产生结果的节拍之间的拍数。这样,以上面的示例流水线为例,浮点加的拍数为 4,而浮点乘的拍数为 7。为达到高时钟速度,设计者们要在流水线的每一节拍设置较少的逻辑层,这就使复杂的操作所需的拍数增多了。可以看出,提高时钟速度的代价是操作延迟的增加。

图 12.17 所示的示例流水线结构,允许至多 4 个浮点加、7 个浮点/整数乘和一个浮点除操作同时进行。图 12.18 所示为扩展了的图 12.16 的流水线,在图 12.18 中重复间隔的实现为:增加一些额外的流水线节拍,而这些节拍可被额外的流水线寄存器分离出来。因为各单元彼此独立,所以我们将各节拍分别命名。需要多个时钟周期的流水线节拍,如除单元的节拍,被进一步细分,以指明这些节拍的延迟。但是因为它们不是完整的节拍,只能有一个操作是活跃的,也可以使用类似于本章中前面使用过的示意图来表示流水线结构。如图 12.19 所示为一组独立的浮点操作,浮点 load 和浮点 store 操作。自然地,浮点操作的长延迟导致了先写后读相关和其造成的阻塞的增加,这些我们将在本章的稍后部分涉及到。

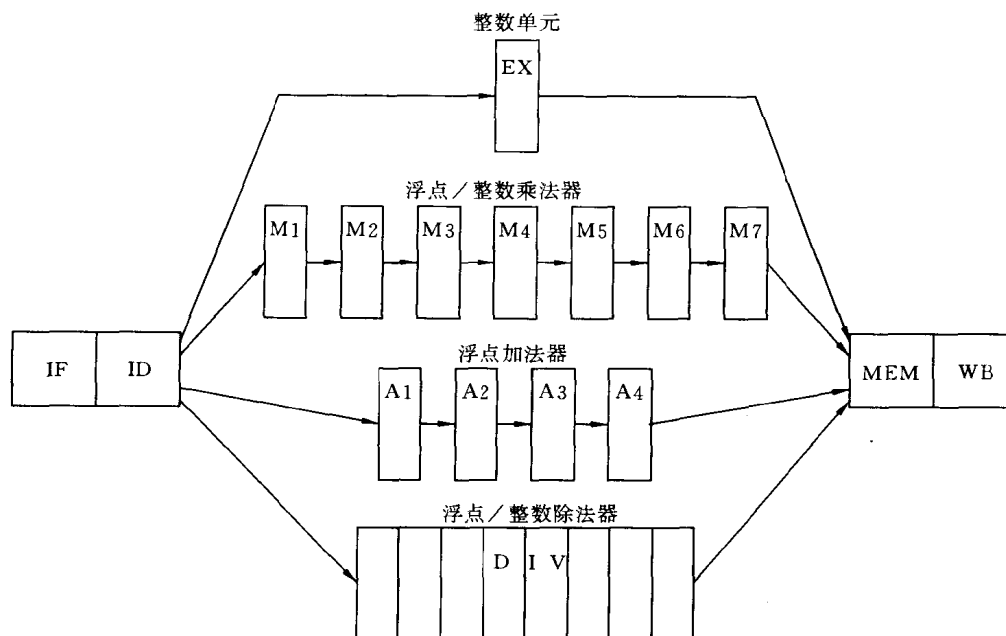


图 12.18 一个支持多个浮点操作同时进行的流水线。浮点乘法器和加法器是全流水化的，并且深度分别为 7 个节拍和 4 个节拍。浮点除法器没有流水化，但是需要 25 个时钟周期来完成。在指令中，在浮点操作的发射和不产生先写后读阻塞的情况下使用该操作的结果这二者之间的延迟，取决于在执行节拍中所花费的周期数。例如，浮点加之后的第四条指令就可以使用该浮点加的结果。对整数 ALU 操作来说，执行流水线的深度总是 1，下一条指令总是可以使用它的运算结果。浮点 load 和整数 load 都是在 MEM 步完成，这就意味着存储器必须在一个单独的时钟周期内提供或者 32 位或者 64 位的数据

MULTD	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM
ADDD		IF	ID	A1	A2	A3	A4	MEM	WB	
LD			IF	ID	EX	MEM	WB			
SD				IF	ID	EX	MEM	WB		

图 12.19 一组独立的浮点操作的流水线时序。用斜体印刷的节拍表明它需要数据。黑体印刷的节拍表明结果可被使用。浮点 store 像整数 store 一样，在不同的时间两次使用它的源操作数。

要理解图 12.18 中的流水线结构，需要介绍一下附加的流水线寄存器（如 A1/A2，A2/A3，A3/A4）和与这些寄存器的连接。ID/EX 寄存器必须扩展到能把 ID 与 EX、DIV、M1、A1 连接起来，我们可以将寄存器中与下一节拍中的某一个相关联的端口用符号 ID/EX、ID/DIV、ID/M1 和 ID/A1 来标识。ID 和所有节拍间的流水线寄存器必须被认为是逻辑上独立的寄存器，并且事实上可以被实现为独立的寄存器。由于在同一时间内，一个流水线节拍内只能有一个操作，那么控制信息就可以在节拍的开头部分与寄存器联系

起来。

12.2.3.1 在长延迟的流水线中的相关和专用通路

对于图 12.18 中所示的那种流水线来说,相关检测和设置专用通路有一些新的特点:

(1) 因为除法单元不是全流水化的,就会出现结构相关,这些相关需要被检测到,并且正在执行的指令需要被阻塞。

(2) 因为各种指令的运行时间不同,所以一个周期内所需的寄存器写的次数可以大于 1。

(3) 因为指令不再按顺序到达 WB,所以可能出现输出相关。因为寄存器读操作总在 ID 节拍进行,所以不存在 WAR 相关。

(4) 指令可以按照与它们被发射时不同的顺序执行完毕,这将引起一些意外的问题。

(5) 由于操作的长延迟,相关所引起的阻塞将更频繁地出现。

操作的长延迟所引起的阻塞次数的增加基本上与整数流水线相同。在描述浮点流水线所产生的新问题并给出解决方案之前,让我们先看看先写后读相关的潜在影响。图 12.20 所示为一个典型的浮点(操作)代码序列和其造成的阻塞,在这一节的末尾,我们将在 SPEC 子集上测试这个浮点流水线的性能。

时 钟 周 期 数															
指令	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LD F4,0(R2)	IF	ID	EX	MEM	WB										
MULTD F0,F4,F6	IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB			
ADDD F2,F0,F8				IF	ID	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM
SD F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	stall	stall	stall	MEM

图 12.20 用于说明有先写后读相关引起的阻塞的一段典型浮点运算代码序列。相对于深度较浅的整数流水线,较长的浮点流水线使得阻塞出现的频率增加。在假设流水线是设置专用通路(forwarding)的前提下,这一序列中的每一条指令都与前一条相关,并且在数据可用时,马上就继续执行。注意 store 双精度数指令(SD)与 ADDD 指令同时执行它们的 MEM 周期,但是,因为这两条指令在不同的流水线,并且 SD 已经有了数据,这个时序将能够正常工作

现在看看上面的(2),(3)两条所描述的由写操作所引起的问题。如果我们假设浮点寄存器堆有一个写端口,那么浮点操作序列可能引起寄存器写端口的冲突,正如在浮点操作的同时进行浮点 load 时一样。例如,考虑图 12.21 所示的流水线序列,在周期 11,全部三条指令都到达,并且都要写寄存器堆,在只有一个寄存器写端口的情况下,机器必须使指令的执行串行化。单个寄存器端口意味着一个结构相关。我们可以通过增加写端口数来解决这个问题,但是由于额外的写端口很少被用到,这种方案就显得缺乏吸引力,这是因为绝大多数情况下写端口号只需要一个。相反地,我们选择的方案是检测对写端口的访问,并把多个同时的写操作看作结构相关。

时钟周期数											
指令	1	2	3	4	5	6	7	8	9	10	11
MULTD F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F8,0(R ₂)							IF	ID	EX	MEM	WB

图 12.21 如周期 11 所示,三条指令试图同时完成写浮点寄存器堆的操作,但这并不是最糟糕的情况,因为早一些的一个浮点单元中的除法也可以在同一周期内完成

有两种办法来实现这种互锁,第一种是在 ID 节拍跟踪写端口的使用,并且像对待其它结构相关一样,在指令发射前就阻塞它。跟踪写端口的使用是通过一个移位寄存器来实现的,它指明已经发射的那些指令将在何时使用寄存器堆。如果 ID 中的指令需要与已经发射的指令同时使用寄存器堆, ID 中的指令就阻塞一个周期。在每个时钟周期内,预约(reservation)寄存器都会移位一位。这种实现有一个优点:它维持了只在 ID 节拍阻塞指令的特性。代价是增加了移位寄存器和写冲突(判断)逻辑。我们这一节中,都假定使用这种方案。

另一种方案是在冲突的指令试图进入 MEM 节拍时再阻塞它。如果我们在产生冲突的指令在进入 MEM 节拍前阻塞它们,就可以选择被阻塞的指令。一个简单的、但有时并不是最优的启发式算法是给那些有长延迟的单元以优先权,因为它们是最容易导致其它指令被阻塞,从而产生先写后读相关的。这种方案的优点是直到进入容易检测冲突的 MEM 节拍以前,我们都不需要检测冲突。缺点是使流水线的控制变得复杂,因为可能在两个不同的地方产生阻塞,这一进入 MEM 节拍前的阻塞将引起 EX、A4 或 M7 等节拍被填满,这就可能会迫使阻塞在流水线中向后传播。

另外的一个问题是可能会有输出相关。考虑到图 12.21 中的例子,我们会看到这种情况是存在的。如果 LD 指令提前一个周期发射,并且目标操作数是 F2,那么它将产生一个输出相关,因为它比 ADDD 指令早一个周期写 F2。注意只有 ADDD 指令的结果在没有任何指令使用它的情况下被覆盖时,这种相关才会出现。如果在 ADDD 和 LD 之间有对 F2 的作用,流水线就需要因为有先写后读相关而被阻塞。并且 LD 要等到 ADDD 执行完毕后才发射。对我们的例子来说,输出相关只有在执行一条无用指令时才会出现。但是,我们仍然必须检测到它,并且确保当我们结束时,LD 的结果在 F2 中出现。

有两种可能的方案可以处理这种输出相关。第一种方案是延迟 load 指令,直到 ADDD 进入 MEM。第二种方案是通过检测相关和改变控制使 ADDD 并不写其结果。然后,LD 就能立刻发射了。因为这种相关极少出现,所以两种方案都很有效。你可以挑选容易些的来实现。在两者中的任一情况下,在 LD 的 ID 节拍相关就会被检测到,然后阻塞 LD 或使 ADDD 成为一个空操作(no-op)就容易了。困难的情形是去检测 LD 可能在 ADDD

之前结束。因为这需要知道流水线的长度和 ADDD 现在的位置。幸运的是,这个代码序列(两个写操作,中间没有读操作)很少见。所以我们采用一个简单的解决方案:如果一条 ID 节拍中的指令试图与已经发射的指令同时写一个寄存器,就不将其发射到 EX。首先,让我们将浮点流水线中的相关和发射逻辑的实现细节(pieces)结合到一起。

在检测可能的相关时,我们既要考虑浮点指令之间的相关,又要考虑浮点指令与整数指令之间的相关。除了浮点 load 和 store 指令和浮点-整数寄存器移动指令,浮点寄存器和整数寄存器是独立的。所有的整数指令都在整数寄存器进行操作,而所有的浮点操作只在它们自己的寄存器上进行。这样,在检测浮点指令和整数指令间的相关时,我们只需要考虑浮点 load/store 和浮点寄存器移动指令。这种流水线控制的简化是整数数据和浮点数据具有不同的寄存器堆的做法所附带而来的优点。(主要的优点是使寄存器数加倍,而每个寄存器堆并未增大,并且在不增加寄存器端口的情况下,增加了带宽。主要的缺点是:除了需要一个额外的寄存器堆外,还有偶尔出现在两个寄存器堆之间的移动数据而带来的较小的代价。)假定流水线在 ID 节拍检测所有的相关,那么,在一条指令发射前,必须完成如下三种检查:

(1) 检查结构相关:等到所需的运算单元都不忙时(在 DLX 流水线中,只有除法需要),并且确保寄存器写端口在需要时可用。

(2) 检查先写后读数据相关:等待源寄存器不再作为挂起的目的寄存器存在于流水线寄存器中(当指令需要结果时,这样的寄存器是不可用的)。在此必须作许多检查,这取决于源指令和目的指令。源指令决定了结果何时可用。目的指令决定了何时需要它的值。例如,ID 节拍中的指令是一个源寄存器为 F2 的浮点操作,那么 F2 就不能作为 ID/A1, A1/A2, A2/A3 的目的寄存器,这正符合当 ID 中的指令需要结果时,浮点加指令还不能结束的情况,(ID/A1 是 ID 对 A1 中的指令输出寄存器的接口)。如果我们试图允许除法的最后几个周期重叠的话,对除法的处理就有些麻烦,因为我们必须在除法将要结束的时候,特殊处理这种情况。实际上,出于简化发射测试的目的,设计者大概会忽略这种优化。

(3) 检查输出相关:决定是否有 A1...A4, D, M1...M7 中的指令与这条指令有相同的目的寄存器。如果有,就在 ID 节拍中阻塞这条指令的发射。

尽管由于多周期的浮点操作,相关检测变得复杂了,但是其概念与 DLX 整数流水线是相同的。专用通路逻辑也是如此,专用通路逻辑可以通过检查在 EX/MEM, A4/MEM, M7/MEM, D/MEM, 或 MEM/WB 寄存器中的目的寄存器是否是一条浮点指令的源寄存器。如果是的话,相应的输入多路器就被使能,以使它选择专用通路的数据。

多周期浮点操作也给我们的异常(exception)处理机制带来了问题。我们将在下面给予处理。

12.2.3.2 维持精确的异常处理

由运行时间长的指令引起的另外一个问题可以由下面的这段代码序列来说明:

```
DIVF F0, F2, F4
ADDF F10, F10, F8
SUBF F12, F12, F14
```

这段代码看起来很简单,各条指令间没有依赖关系,但是由于一条先发射的指令可能是在后发射的指令之后执行完毕,问题就产生了。在这个例子中,我们可以预期到,ADDF 和 SUBF 指令在 DIVF 指令之前执行完毕。这种情况被称为乱序结束(out-of-order completion),在有长运行时间的指令存在的流水线中普遍存在。可是,既然相关检测防止了任何有依赖性的指令间的互相干扰,为什么乱序结束还会成为一个问题呢?设想 SUBF 在 ADDF 已经完成,但是 DIVF 结束前的一点引起了一个浮点算术异常,结果将是不精确的异常,这正是我们试图要避免的。这个问题看起来可以像对待整数流水线那样,通过清空浮点流水线来解决,但是,异常可能在一个不能采用这种方法的位置出现。例如,如果 DIVF 决定在加结束之后产生一个浮点算术异常,我们即使在硬件级别上都不能获得一个精确的异常。事实上,由于 ADDF 破坏了它的一个操作数,我们即使在软件的帮助下,也不能恢复到执行 DIVF 之前的状态。

这个问题的产生是由于指令以与它们发射时不同的顺序完成。有四种可能的方案解决乱序结束。一是忽略这个问题,满足于不精确的异常。这种方案在 60 年代和 70 年早期被使用。现在某些巨型机仍在使用。在这些机器中,某些类的异常被禁止,或在硬件级别上处理,而不是中断流水线。对于大多数现在制造的计算机来说,很难再采用这种策略,因为虚存和 IEEE 浮点标准等特性要求通过软硬件结合产生的精确的异常。正如我们前面提到,近期机器通过引入两种异常模式解决了这个问题:一个快速、但可能是不精确的模式,和一个慢速、但精确的模式。慢速、精确的模式或通过一个模式开关来实现,或通过插入一个显式的指令测试浮点异常来实现。在这种情况下,浮点流水线中所允许的重叠和重定序(reordering)的数目被限制,以使在同一时间内只有一条浮点指令是活跃的。这种方案被用于 DEC Alpha 21064 和 21164,IBM Power-1 和 Power-2,MIPS R8000。

第二种方案缓存操作的结果直到所有在该操作之前发射的操作都执行完毕。有些机器的确使用这种方案,但是当不同操作间的允许时间相差很大时,由于要缓存的结果变得很大,这种方案就变得代价过高了。另外,在等待较长的指令的时候,队列中的结果必须被 bypass,以便继续发射后面的指令。这需要大量的比较器和一个非常大的多路选择器。

在这个基本的方案的基础上,有两个不同的变种。第一种是在 CYBER 180/190 中使用的历史表(history list)。历史表中记录寄存器的原始值。当异常出现,并且状态必须要卷回到一些乱序结束的指令之前的时候,寄存器的原始值就可以被从历史表中读出并得到恢复。类似的技术在 VAX 之类的机器中用于自动增量和自动减量寻址。另一种方案是由 J. Smith 和 A. Pleszkun 提出来的未来表(future file),这种方案保存寄存器的新值,当所有该指令之前的指令都执行完毕之后,主寄存器堆根据未来表更新。

第三种方案是允许异常不十分精确,但是要保存足够多的信息以使陷入(trap)处理例程为这个异常建立一个精确的序列,这意味着要知道它们的流水线有哪些操作以及其 PC 值,然后在处理完异常后,由软件完成那些在刚完成的指令之前的指令,之后重新启动指令序列。考虑如下的最坏情况的代码序列:

指令 1: 一个长时间运行的、最后中断了运行指令。

指令 2...指令 $n-1$: 一系列未完成的指令。

指令 n : 一条执行完的指令。

在给定了流水线中各指令的 PC 和异常返回 PC 后,软件就能确定指令 1 和指令 n 的状态了。因为指令 n 已经完成,我们希望从指令 $n+1$ 开始重启执行。在处理异常后,软件必须模拟指令 1...指令 $n-1$ 的执行。然后我们就可以从异常返回,再从指令 $n+1$ 重启。处理函数正确执行这些指令的复杂性是这种方案的主要困难,简单 DLX 类型的流水线有一个重要的简化:如果指令 2...指令 n 都是整数指令,那么我们如果知道指令 n 执行完了,那么也就知道指令 2...指令 $n-1$ 也已经执行完了。这样,只有浮点操作需要被处理。为使这个方案易于管理,执行时所允许重叠的浮点指令数就要加以限制。例如,如果我们只重叠两条指令,那么只有发生中断的指令需要有软件完成执行。如果浮点流水线很深或者有很多的浮点运算单元的话,这种限制将减少潜在的吞吐率。这种方案被用于 SPARC 结构,以允许浮点操作和整数操作的重叠。

最后一种方案是一种混合的方案,它只有在确信正要发射的指令之前的那些指令都会不发生异常地执行完毕的时候,才允许该指令继续发射。这就确保了当异常出现时,在

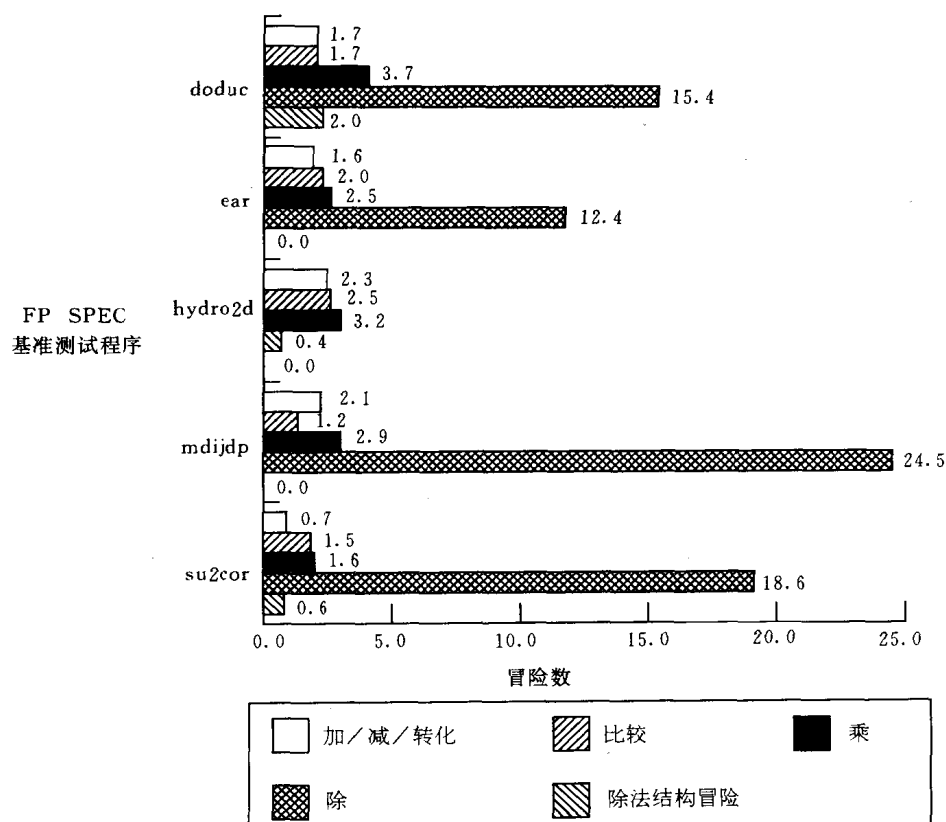


图 12.22 每种主要类型的浮点操作的每个操作的阻塞数(阻塞/操作)。除了除法结构相关,这些数据不依赖于操作出现的频率,而是只依赖于它的延迟和结果被使用之前的时钟周期数,先写后读相关引起的阻塞数大致反映了浮点单元的延迟。例如,每个浮点加、减、转换的平均阻塞数为 1.7 个周期,或是延迟(三个周期)的 56%,同样地,乘和除的平均阻塞数是 2.8 和 14.2,是各自延迟的 46%和 59%。由于除法出现的频率很低,所以除法的结构相关很少见

中断的指令之后的那些指令都不会结束。而它之前的所有指令都已经结束。这意味着有时要靠阻塞机器来维持精确的异常。为使这种方案工作,浮点运算单元必须在 EX 节拍的前期(在 DLX 流水线的前三个周期中)判断是否可能会出现异常,以防止后面的指令继续执行。这种方案被用于 MIPS R2000/R3000, R4000, 和 Intel Pentium。

12.2.3.3 浮点流水线的性能

图 12.18 所示的 DLX 流水线会产生除法单元的结构性阻塞和由先写后读相关引起的阻塞(还可能产生输出阻塞,但在实际中很少出现),图 12.22 所示为每条指令需要阻塞的平均周期数。正如我们预期地,每个操作的阻塞的周期数与浮点操作的延迟相关,为运算单元的延迟的 46%到 59% 不等。

图 12.23 用我们使用的五个浮点 SPEC 指标,给出了整数阻塞和浮点阻塞的分类数据。有四类阻塞:浮点结果性阻塞、浮点比较阻塞、load 延迟和分支延迟、浮点结构性延迟。编译器在调度分支延迟之前,调度 load 延迟和浮点延迟。每条指令的总阻塞数从 0.65

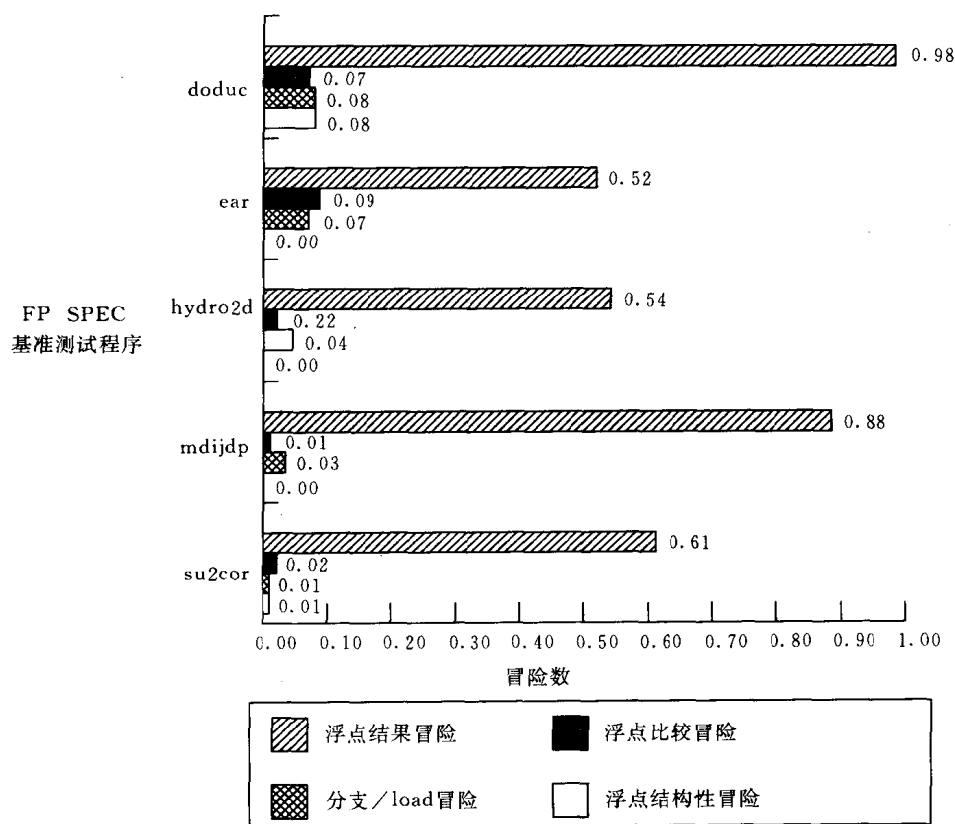


图 12.23 五种 SPEC 指标下 DLX 流水线上出现的阻塞。每条指令的总阻塞数从 su2cor 的 0.65 到 doduc 的 1.21, 平均为 0.87。浮点结果阻塞在所有情况下都占统治地位, 平均每条指令 0.71 个阻塞, 或为阻塞周期总数的 82%。比较操作的每个操作产生平均为 0.1 的阻塞, 是第二大来源。对 doduc 来讲, 除法结构相关是唯一重要的因素

到 1.21 不等。

12.3 实验环境与内容

12.3.1 实验环境

本实验使用 WinDLX 模拟器。该模拟器可以装入 DLX 汇编语言程序,然后单步、设断点或是连续执行该程序。CPU 的寄存器、流水线、I/O 和存储器都可以用图形表示出来。模拟器还提供了对流水线操作的统计功能。该模拟器对理解流水线和 RISC 处理器的其它方面的特点非常有帮助。

WinDLX 由 Professor Herbert Grünbacher, University of Technology Vienna, Inst. für Technische Informatik 开发,可以在互连网上通过 <ftp://mkp.com/pub/dlx/windlx.exe> 得到。

WinDLX 要求的硬件平台是 IBM-PC 兼容机。WinDLX 是一个 Windows 应用程序,运行 DOS 3.3 以上和 Windows 3.0 以上的操作系统。

WinDLX 软件包中带有说明文件,供安装程序时使用。软件包还有 WinDLX 教程和联机帮助,可以供读者进一步了解模拟器的使用方法和 DLX 处理器的原理。读者在进行实验前应该仔细阅读这些文档。

12.3.2 实验内容

12.3.2.1 使用 WINDLX 模拟器,对 Fact.s(WINDLX 附带的例子)作如下分析

(1) 观察程序中出现的的数据相关、控制相关和结构相关现象。指出程序中出现上述现象的指令。

(2) 考察设置相关专用通路技术对于减少相关现象的作用。

(3) 考察增加浮点运算部件对性能的影响。

(4) 观察转移指令在转移成功和不成功时的流水线开销。

* 除(3)外,浮点加、乘、除单元都只有一个

* * 本题中所有浮点运算单元的延时均为 4 个周期。

12.3.2.2 用 DLX 汇编语言编写矩阵乘程序,并对该程序做如下分析

(1) 重复 12.3.2.1 节中(1),(2),(3)中的工作。

(2) 使用循环展开(loop unrolling)手工优化程序。

(3) 对(2)所得的程序,重复 12.3.2.1 节中 3)的工作。

(4) 对(2)所得的程序,把所有浮点单元的延迟改为 8 个时钟周期,重复 12.3.2.1 中(3)的工作。

(5) 计算该程序所需的主存带宽,当主存带宽为 40MB 时,所能支持的最大处理器主频是多少?

* 为简单起见,可以固定矩阵的大小,如 20×20 ,可以不赋初值,不输出计算结果,目的是考察矩阵乘循环的指令序列

12.3.2.3 用 Intel X86 汇编语言编写矩阵乘程序

(1) 尽量优化你的程序,比较 Intel X86 汇编语言的程序与 DLX 汇编程序的风格的不同。

(2) 比较 X86 汇编语言程序与 DLX 汇编程序的大小。

12.3.3 实验总结

根据实验,总结流水线会遇到的问题和为解决这些问题所采用的各种技术的作用,并谈谈你对流水线技术的认识。

参 考 文 献

- [1] 郑纬民,陈修环,石岩,汤志忠. 计算机系统结构. 北京:清华大学出版社,1992 年
- [2] Kai Hwang 著,王鼎兴,沈美明,郑纬民,温冬婵译. 高等计算机系统结构 并行性 可扩展性 可编程性. 北京:清华大学出版社,1995 年
- [3] 李学干,苏东庄. 计算机系统结构. 西安:西安电子科技大学出版社,1991 年
- [4] 李勇,刘恩林. 计算机体系结构. 长沙:国防科技大学出版社,1988 年
- [5] 黄铠著,金兰等译. 计算机结构与并行处理. 北京:科学出版社,1990
- [6] 金兰. 计算机组织与结构. 北京:高等教育出版社,1986 年
- [7] 陆鑫达. 计算机系统结构. 北京:高等教育出版社,1996 年
- [8] 王鼎兴,陈国良. 互连网络结构分析. 北京:科学出版社,1990 年
- [9] Patterson D A, Hennessy J L. Computer Architecture: A Quantitative Approach 2 Ed. San Francisco: Morgan Kaufmann Publishers. 1995
- [10] Stone H S. High-performance Computer Architecture. Addison-wesley, 1987
- [11] Richard Y Kain. Advanced Computer Architecture A Systems Design Approach. Prentice-Hall, Inc. ,1996
- [12] James H Herzog. Design and Organization of Computer Structures. Franklin, Beedle & Associates, Inc. ,1996